Re-usability and Robustness of Python and Java programs

Debdas Paul
Department of Computer Science and Engineering
Lehigh University
Bethlehem, USA
Email: dep411@lehigh.edu

Sourabh Vartak
Department of Computer Science and Engineering
Lehigh University
Bethlehem, USA
Email: sjv211@lehigh.edu

***Abstract*—“Re-usability” is one of the fundamental aspects of software development. Re-usability of code enables us to add new functionality with minimum modifications. A module or class which is reusable reduces the implementation time and cost of maintenance. In this paper we investigate and compare the re-usability of Python and Java programs using graph theoretic approach. We define a metric to quantify re-usability. Using the metric, we show that Python projects are more reusable than Java, which supports the design goal of Python. Apart from re-usability, we also address the question of “Robustness” for software programs. We cannot test robustness by removing nodes or edges from software dependency graph because removal of any nodes results into software crash. This makes the task more challenging. Therefore, we redefine robustness for software programs. We also define a metric to quantify robustness and compare Python and Java projects using the metric. We show that, Java programs are more robust than that of Python against random failure.**

***Keywords*-Python; Java; Graph Theory; Re-usability; Robustness.**

I. Project Overview

Re-usability of a segment of source code measures likelihood of its future use with slight modification or with no modification at all (http://en.wikipedia.org/wiki/Reusability). A module or class which is reusable reduces the implementation time and cost of maintenance. Therefore, a software must be designed in such way so as to increase its re-usability. Object-oriented languages like Java have been designed to promote re-usability [1]. Now, the interesting scenario happens in case of languages like Python. Python supports multiple programming paradigms like object-oriented, imperative and, to some extent, functional programming. Moreover, Python programming language design emphasizes on code readability and code readability increases re-usability. Now, the question is whether Python code is more reusable due to better readability. We propose to find the answer by comparing structural properties of dependency graph of Java and Python modules. If a module is less dependent on other modules, then that module is said to be more reusable. In terms of dependency graph, the greater the out-degree than in-degree of a module, the more reusable the module is. If Python comes out to be more re-usable than Java, then it also has implications on writing readable codes.

Recent studies show that geometry[1] of object oriented programming is “scale-free” in nature [2], [3]. A scale-free network is a network whose degree distribution follows a power law asymptotically (http://en.wikipedia.org/wiki/Scale-free_network). Mathematically, if a fraction of node $P(k)$ of nodes in the network having k connections to other nodes goes for large values of k as

$$P(k) \sim ck^{-\gamma}$$

, where where c is a normalization constant and γ is a parameter whose value is typically in the range $2 < \gamma < 3$. The value of γ may lie outside the bounds. For example, Barabasi-Albert model of graph follows power law with $\gamma = 3$. Random graphs do not follow power law [4]. One of the important properties of scale-free network is that, it is robust against random failure [4]. Random failure happens when nodes other than the hubs are attacked. But, in case of object-oriented software, we need to redefine “Robustness” because if any node/module, whether it is hub or not, fails then the entire system fails. Therefore, instead of removing a node/module here, we can think about the extend of effect due to some external modification in that node/module. The longer the effect propagates, the less robust is the network with respect to that node/module. Note that, scale-free networks are robust against random failure, but not against targeted failure (where the hubs are directly attacked) [4].

To our knowledge, robustness for software programs has not quantified yet. Therefore, first we propose to investigate the dependency graph of Python modules in large open source project(s) to see whether it is scale-free or not and then define a metric for measuring the robustness of the same. Using our metric, we show that Java projects are more robust than that of Python against random failure.

II. Materials and Methods

A. Open Source projects

We use four open source projects. Two of them are from Java and other two are from Python. The projects

[1]Here the geometry means the network of dependency between classes and modules

are comparable in terms of task they perform. For Java we choose *Google Web Toolkit* and *BioJava*. For Python we choose, *Pyjamas* and *BioPython*. Below is the short description of each of these.

1) Google Web Toolkit: It is a open source Java software development framework for writing AJAX based applications like Google Maps and Gmail (http://code.google.com/webtoolkit/).

2) Pyjamas: It is Rich Application Development framework for both Web ad Desktop. It contains a Python-to-Javascript compiler which is an AJAX framework and a Widget Set API (http://pyjs.org/).

3) BioJava: BioJava is an open-source project in Java for processing biological data. It provides analytical and statistical routines, parsers for common file formats and allows the manipulation of sequences and 3-D structures (http://biojava.org/wiki/Main_Page)

4) BioPython: BioPython is an open source Python framework for biological data analysis (http://biopython.org/wiki/Main_Page).

B. Softwares for creating dependency graph

1) For python: We use "snakefood" (http://furius.ca/snakefood/) for creating dependency graph for Python projects. We choose this software based on the following reasons: *1)* It uses Abstract Syntax Trees for parsing the Python files which is reliable and easy to use. *2)* It does not load modules while calculating dependency graph. *3)* It finds all the files in a directory recursively. No need for explicit declaration of files. *4)* It can follow dependencies by enabling "–follow" option in the command line. *(5)* It supports UNIX command line features. We can easily join the commands using pipes.

2) For Java: We use "JDep-Grapher" (https://github.com/Kdecherf/jdep-grapher) for creating dependency graph for Java projects. We choose this software based on the following reasons: *1)* It is a simple Bash tool for generating dependency graphs. *2)* Given a source code directory as input, it recursively finds and parses all Java files in all the sub-directories. *3)* Since it is a simple shell script, it supports UNIX command line features like redirection to output file.

C. Software for analysis

We use "NetwokX" module for analysis of dependency graphs. NetworkX is a powerful module in Python for creation, manipulation and analysis of complex networks (http://networkx.lanl.gov/).

D. Methods

We use the following steps for analysis of re-usability and robustness in Java and Python projects.

1) Generation of Python dependency graphs: We download all the six versions so far developed for Pyjamas. For BioPython, we choose six versions uniformly over all versions so that, we can get significant variations. Below is the table which shows the version and its size.

Version	Number of Python files
0.3	182
0.4	254
0.5	384
0.6	586
0.7	874
0.8	1236

Table I: Size of versions of Pyjamas

Version	Number of Python files
1.10	417
1.30	527
1.44	565
1.48	565
1.55	467
1.57	436

Table II: Size of versions of BioPython

We generate dependency graph using the following command line for Python projects:

```
sfood --ignore-unused --follow <project>
| sfood-graph | dot > <project>.dot
```

"–ignore-unused" option eliminates dependencies motivated by symbols imported but not used. "–follow" option allow sfood to follow dependencies recursively. At the end of this pipeline we get a .dot file. We then convert the .dot file to Python graph data structure using NetworkX for further analysis.

2) Generation of Java dependency graphs: We download six recent versions of the Google Web Toolkit. Similarly, for BioJava, we choose six versions uniformly over all versions so that, we can get significant variations. For generating dependency graph for Java projects, we use the following command line:

```
jdep-grapher.sh <project.dot>
<project_src_dir>
```

Version	Number of Java files
1.4.62	468
1.7.1	960
2.0.0	1230
2.1.0	1943
2.2.0	2355
2.4.0	3041

Table III: Size of versions of Google Web Toolkit

We have modified the jdep-grapher shell script so that it does not generate the PNG image file for the dependency

Version	Number of Java files
1.6.1	1405
1.7.0	1462
1.7.1	1476
1.8.1	1467
3.0.1	526
3.0.2	600

Table IV: Size of versions of BioJava

graph. For large dependency graphs, as in our case, the JDep-grapher tool becomes unresponsive when generating the PNG image file. So, using modified jdep-grapher script, we only create the DOT file which will be used in our analysis.

III. Results

A. Reusability

We know that a module or code is more reusable in term dependency graph if out-degree of a node is much more higher than in-degree of that node. Ideally, more the amount of 0 in-degree nodes in a dependency graph of a module, more the percentage of reusable code in that module. We can define a metric of "AReuse"

$$AReuse_X = \frac{\sum_{i=1}^{V} N_i}{\sum_{i=1}^{V} TotN_i} \qquad (1)$$

, where N_i is the total number of nodes in a version i for which out-degree is greater than in-degree; $TotN_i$ is the total number of nodes in version i and X is the project. If $AReuse_X$ is greater than $AResue_Y$, then we can say that X contains more reusable codes than Y. Below is the $AReuse$ values for Java and Python projects.

Project	$AReuse$
GWT	0.09
PyJamas	0.47
BioJava	0.11
BioPython	0.59

Table V: Average re-usability of Python and Java projects. It is clear from the above data that Python has more re-usability than Java

From Table V, it is clear that Python projects are much more reusable than Java. Python is designed to be more readable and also "As an object-oriented language, Python aims to encourage the creation of reusable code" (http://www.python.org/doc/essays/foreword/). This, we verify the greater re-usability of Python code.

B. Robustness

For software programs we have to define the robustness in terms of extent of the effect of modifiability. There are two types of robustness: robustness against random failure and robustness against targeted failure/attack. Scale-free networks are generally robust against random failure because of large number of low degree nodes. Random graphs are more robust against targeted failure. From analysis, we get that the in-degree and out-degree distribution follow power law asymptotically, which means the over all dependency graph is a directed scale-free network. Below is an example of BioJava in-degree distribution which follows power law.

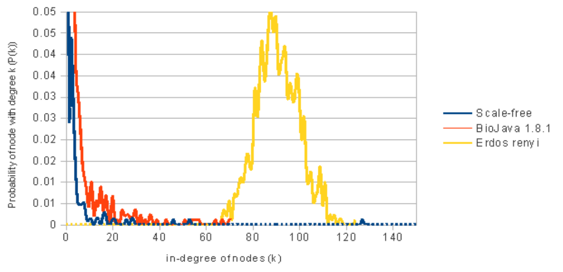

Figure 1: Above figure shows the difference between scale-free networks (in blue and red) and a random graph (Erdos-Renyi, in yellow). Random graph does not follow power law degree distribution asymptotically. Red line represents the in-degree distribution of BioJava network which is scale-free. The same holds for other Python and Java projects too.

To prove the scale-freeness, we generate 100 random scale free directed graphs using NetworkX for each of the projects. We take the average number of nodes across all the versions of a project for generating the corresponding random scale free graph. We investigate the tail of the graphs and find the average value of exponent γ corresponding to each project. Note that, this γ is not the actual γ for a project. We calculate the actual γ separately for each projects.

Scale free graph for corresponding project	Number of nodes in each of the 100 scale free graphs	γ_{avg}^{in}	γ_{avg}^{out}
GWT	1060	1.755	2.078
PyJamas	745	1.733	2.044
BioJava	844	1.741	2.055
BioPython	958	1.75	2.076

Table VI: Power law statistics for randomly generated scale free directed graphs

Versions for BioJava	Number of nodes	γ^{in}	γ^{out}
biojava-1.6.1	890	1.9	2
biojava-1.7.0	967	1.9	1.9
biojava-1.7.1	968	1.9	1.9
biojava-1.8.1	907	1.8	1.9
biojava-3.0.1	635	2	1.9
biojava-3.0.2	699	2	1.9

Table VII: Power law statistics for BioJava

Comparing Table VI with Tables VII, VIII, IX and X, we can conclude that the projects are scale free nature. If a network is scale-free, it is robust against random failure

Versions for BioPython	Number of nodes	γ^{in}	γ^{out}
bio1.10	737	1.9	2.4
bio1.30	854	1.9	2.5
bio1.44	894	1.9	2.5
bio1.48	894	1.9	2.5
bio1.55	1184	1.9	2.1
bio1.57	1187	1.9	2.1

Table VIII: Power law statistics for BioPython

Versions for GWT	Number of nodes	γ^{in}	γ^{out}
gwt-user-1.4.62	276	2.2	1.9
gwt-user-1.7.1	561	2.1	1.8
gwt-user-2.0.0	964	2.0	2.0
gwt-user-2.1.0	1335	2.3	2.0
gwt-user-2.2.0	1432	2.3	2.0
gwt-user-2.4.0	1793	2.3	2.0

Table IX: Power law statistics for Google Web Toolkit

Versions for Pyjamas	Number of nodes	γ^{in}	γ^{out}
py0.3	337	1.9	2.1
py0.4	395	1.9	2.1
py0.5	544	1.9	2.2
py0.6	778	1.9	2.6
py0.7	1064	2.0	2.6
py0.8	1357	2.0	2.5

Table X: Power law statistics for Pyjamas

[5]. Now the question is which one is more robust against random failure : Java or Python. Here, we create a new metric : Average Robustness or AR

$$AR_X = \frac{\sum_{i=1}^{V} \frac{N_{1_{in}}^{v_i}}{N_{1_{out}}^{v_i}}}{V} \tag{2}$$

, where V is the number of versions in X. X may be one of the open source projects. If $AR_X > AR_Y$, X is more robust than Y against random failure. The reason is that, if we choose randomly a node in X, the probability of getting 1-in degree node is much more higher compare to that in version Y and any modification on that node does not propagate to any other nodes. Therefore, extent of random modifiability in case of version X is less than Y. Another question one might ask why don't we consider 0-in-degree or 0 out-degree nodes. The reason is that a hub node can be a 0-in-degree or 0-out-degree node. But, here we do not consider hubs as we are considering robustness against random failure. From the Table XI, it is clear that robustness of Java code against random failure is much more higher than that of Python.

Project	AR
GWT	126.7
PyJamas	2.0
BioJava	206.5
BioPython	2.3

Table XI: Measurement of average robustness against random failure/modification

IV. Conclusion

In this work, we define measures for re-usability and robustness of software programs. We verify it for Python and Java projects. We see Python projects contain more reusable codes and it supports the design principle of Python. Java is more robust against random failures. We still do not know the relation between re-usability and robustness. We keep this for future work.

Acknowledgment

The authors would like to thank Prof. Gang Tan of Department of Computer Science and Engineering, Lehigh University, USA for giving valuable comments while preparing this document. The authors would also like to thank Dr. Alex Potanin, Senior Lecturer in Software Engineering School of Engineering and Computer Science Victoria University of Wellington, New Zealand; Dr Ewan Tempero, Associate Professor in Computer Science Department of Computer Science, University of Auckland for helping in materializing this idea.

References

[1] B. Foote and R. Johnson, "Designing reusable classes," *Journal of Object Oriented Programming*, vol. 1, no. 2, pp. 22–35, 1988.

[2] A. Potanin, J. Noble, M. Frean, and R. Biddle, "Scale-free geometry in oo programs," *Communications of the ACM*, vol. 48, no. 5, pp. 99–103, 2005.

[3] C. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Physical Review E*, vol. 68, no. 4, p. 046116, 2003.

[4] N. Ganguly, A. Deutsch, and A. Mukherjee, *Dynamics On and Of Complex Networks: Applications to Biology, Computer Science, and the Social Sciences*. Birkhauser, 2009.

[5] J. Guillaume, M. Latapy, and C. Magnien, "Comparison of failures and attacks on random and scale-free networks," *Principles of Distributed Systems*, pp. 900–900, 2005.