GetSmells Detection Rules and Algorithms

1 Class Level Smells

1.1 God Class [1]

(ATFD $>$ FEW) $\wedge (WMC \geq VERY\_HIGH) \wedge (TCC < 0.33)$

where
ATFD is Access To Foreign Data,
TCC is Tight Class Cohesion,
WMC is the Weighted Method Count,
(WMC) VERY_HIGH is 47 for Java and 108 for C++,
(AFTD) FEW is 2-5

1.2 Data Class [1]

((NOPA + NOAM $>$ FEW) $\wedge (WMC < HIGH)) \vee ((NOAP + NOAM > MANY) \wedge (WMC < VERY\_HIGH))$

where
NOPA is the Number Of Public Attributes,
NOAM is the Number Of Accessor Methods,
WMC is the Weighted Method Count,
(NOAP+NOAM) FEW and MANY are 2-5,
(WMC) HIGH is 31 for Java and 72 for C++,
(WMC) VERY_HIGH is 47 for Java and 108 for C++,

1.3 Lazy Class [2]

LOC $<$ Q1

where
Q1 is the the first quartile of the distribution of Lines of Codes (LOCs) for all system's classes

1.4 Complex Class [2]

CMC $>$ 1

where
CMC is the Complex Method Count (a complex method is one with McCabe cyclomatic complexity higher than 10)

1.5 Large Class [2]

LOC (of a class) $>$ mean of the system

where
LOC is Lines of Code

1.6 Refused Bequest [2]

LMC $>$ (1/2 * TMC)

where
LMC is the Local Method Count,
TMC is the Total Method Count

1.7 Brain Class [1]

((NBM $> 1) \wedge (LOC >= VERY\_HIGH) \vee (NBM = 1) \wedge (LOC >= 2XVERY\_HIGH) \wedge (WMC >= 2XVERY\_HIGH))$
$\wedge ((WMC >= VERY\_HIGH) \wedge (TCC < HALF))$

where
NBM is the Number of Brain Methods,
LOC is the Lines of Code,
WMC is the Weighted Method Count,
TCC is the Tight Class Cohesion,
(LOC (CLASS) VERY_HIGH is 195 for Java and 360 for C++,
(WMC) VERY_HIGH is 47 for Java and 108 for C++

1.8 Unhealthy Inheritance Hierarchy [3]

We consider an inheritance hierarchy to be problematic if it falls into one of the two cases:
1) Given an inheritance hierarchy containing one parent file, f_{parent}, and one or more children, F_{child} , there exists a child file f_i satisfying depend(f_{parent}, f_i)
2) Given an inheritance hierarchy containing one parent file, f_{parent}, and one or more children, F_{child}, there exists a client f_j of the hierarchy, that depends on both the parent and all its children.

1.9 Hub-Like Dependency [4, 5, 6]

The algorithm to identify this smell is:

• Input: a subgraph of the original dependency graph, where the nodes are classes and the edges represent the dependencies between classes.

• Execution:
1) for all class nodes, calculating the FanIn (number of ingoing dependencies), FanOut (number of outgoing dependencies);
2) calculating the median of FanIn and FanOut metrics of all the classes of the project;
3) checking if the FanIn and FanOut metrics of a class are respectively greater than the FanIn median and FanOut median;
4) checking if the difference between FanIn and FanOut metrics is less than a quarter of the total number of dependencies (sum of FanIn and FanOut) of the class; in this case, the class is a HL smell.

Output: the map of the classes affected by the smell and their relative FanIn and FanOut.

2 Method Level Smells

2.1 Brain Method [1]

((LOC $>$ HIGH(CLASS)/2) $\wedge(CYCLO >= HIGH))\wedge((MAXNESTING >= SEVERAL) \wedge (NOAV > MANY))$

where
LOC is the Lines of Code,
CYCLO is the McCabe's Cyclomatic Complexity,
MAXNESTING is the Maximum Nesting Level,
NOAV is the Number Of Accessed Variables,
(LOC CLASS) HIGH is 130 for Java and 240 for C++,
(CYCLO) HIGH is 0.24 for Java and 0.3 for C++;
(MAXNESTING) SEVERAL is 2-5 NOAV is 7-8

2.2 Feature Envy [7]

LCOM $>=$ High

where
LCOM is the Lack of Cohesion of Methods,
(LCOM) High is 0.725

2.3 Long Parameter List [2]

Number of Parameters (of a method) $>$ mean of system

2.4 Shotgun Surgery [8]

(CM > 10) $\wedge(CC > 5)$

where
CM is the Changing Methods,
CC is the Changing Classes

2.5 Long Method [2]

LOC (of a method) $>$ mean of the system

where
LOC is Lines of Code

3 Package Level Smells

3.1 Unstable Dependency [4, 5, 9]

The algorithm to identify this smell is:

• Input: a subgraph of the original dependency graph, where the nodes are components and the edges represent the afferent coupling (Ca) between components.

• Execution: computing the Instability metric for every component of the graph and update the graph. For every component, checking if the afferent component is less stable; if so, put it in a map with the list of related components less stable.

• Output: the map with every component affected by the smell and the associated components which caused it.

3.2 Cyclic Dependency [4, 5]

The algorithm to identify this smell is:

• Input: a subgraph of the original dependency graph, where the nodes are classes.

• Execution: launching the Depth First Search (DFS) algorithm on the subgraph and collecting every node involved in a cycle in a different list.

• Output: the list of every cycle detected by the DFS algorithm.

References

[1] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media, 2007.

[2] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[3] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *2015 12th Working IEEE/IFIP Conference on Software Architecture.* IEEE, 2015, pp. 51–60.

[4] A. Biaggi, F. A. Fontana, and R. Roveda, "An architectural smells detection tool for c and c++ projects," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* IEEE, 2018, pp. 417–420.

[5] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW).* IEEE, 2017, pp. 282–285.

[6] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2016, pp. 433–437.

[7] M. A. Bigonha, K. Ferreira, P. Souza, B. Sousa, M. Januário, and D. Lima, "The usefulness of software metric thresholds for detection of bad smells and fault prediction," *Information and Software Technology*, vol. 115, pp. 79–92, 2019.

[8] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics.* IEEE, 2015, pp. 44–53.

[9] R. C. Martin, *Agile software development: principles, patterns, and practices.* Prentice Hall, 2002.