The complexity of 4 versions comprised of the following code smells throughout the areas of Complexity, Encapsulation, Coupling, Inheritance,
and Cohesion:

* God class
* SAP Breakers
* Blob Operation
* External Duplication
* Schizophrenic Class
* Data Clumps
* Data Class
* Feature Envy
* Cyclic dependencies
* Unstable Dependencies

==========
Architecture Level smells:
==========
a. Cyclic dependencies

happens when two or more architecture components depend on each other directly or undirectly. The Cyclic dependencies of the 4 version is maintained at a value of 5. This means that the developers may not be aware of this problem since there's no improvements from version 2.8.3 to 2.8.10.Cyclic dependencies is only present with coupling. Because of these cyclic dependencies, this tangles a code and makes it hard for reuse.

b. Unstable dependencies

Is a type of dependency, which could also be cyclical where a component depends on other components that are less stable than itself. There's 1 unstable dependency that's maintained in all 4 versions, that's in coupling. It could also mean that the cyclic dependency value indicated above could also have an unstable dependency.

==========
Class Level smells:
==========
a. God Class

The latest version 2.8.10 has 6 God Classes, version 2.8.9 has 6, version 2.8.4 has 6 and the last version analyzed, 2.8.3 has 5 God classes.Complexity, Encapsulation,

Coupling and Cohesion are tied to the God Class.God Classes are the Class level smells as well as architectural smells. These are classes that have too much responsibility. These classes should be split into smaller classes that have their own responsibility.God classes violate the single-responsibility principle and are hard to unit test and debug.

b. Schizophrenic Class

A schizophrenic class is a class that has disjoint sets of public methods that are used by disjoint sets of client classes. The Schizophrenic classes form V2.8.3 started at 11 then went down to 10 for V2.8.4 but then went up for V2.8.9 and V2.8.10. All of the categories have this type of code smell. It has gotten worse if we look at it starting with the initial version, this also maybe because that the code base is getting bigger, and if the devs didn't take the time to test their code, maintenance by refactoring wasn't possible based on the budget.

c. The presence of Data Classes

The Data classes fluctuate from 33 starting with V2.8.3 to 32 with V2.8.4. The value then goes up to 33 with V2.8.9 and 32 with the latest version. Data classes are only found in encapsulation and cohesion smells.A data class is refers to a class that only contains methods that are getters and setters. These classes don't really have additional functionality and can't operate on their own independently.Data classes prevents lower coupling between the classes. Lower coupling helps to easily maintain classes.

==========
Method level smells:
==========
a. Blob Operation

The Blob operation has decreased from 8 to 7, if we start at version 2.8.3 to 2.8.10. Blob Operations are similar to God classes in a way. They grow larger and larger overtime, becoming more expensive to maintain. Complexity and coupling are the only ones associated with this code smell.

b. External Duplication

The external Duplication for all 4 versions has maintained at 6. External Duplication is associated with Encapsulation and Coupling.

c. Feature Envy

Feature envy has maintained at a level of 5 and is present in Complexity, Encapsulation, Coupling and Cohesion. Feature envy is a code smell describing when

an object accesses fields of another object.Feature envy breaks encapsulation.This is also a coupling code smell, it couples two objects together inappropriately. Since feature envy increases coupling, which increases the likelihood of introducing bugs.

==========
Variable level smells:
==========
a. Data Clumps

Data Clumps have increased from 7 to 11, starting from version 2.8.3 to 2.8.10. Data Clumps is the name given to a group of variables passed around together in a lot of the parts of the program. Because it has increased, the developers may have not been aware of this phenomenon or did not put their code through testing before releasing.

****

Designite:

==========
Architecture Smells:
==========

v2.8.10

Cyclic dependency: 132
God component: 6
Ambiguous interface: 0
Feature concentration: 17
Unstable dependency: 28
Scattered functionality: 0
Dense structure: 1

v2.8.9

Cyclic dependency: 132
God component: 6
Ambiguous interface: 0
Feature concentration: 17
Unstable dependency: 28
Scattered functionality: 0
Dense structure: 1

v2.8.4

Cyclic dependency: 132
God component: 6
Ambiguous interface: 0
Feature concentration: 19
Unstable dependency: 25
Scattered functionality: 0
Dense structure: 1

v2.8.3

Cyclic dependency: 132
God component: 6
Ambiguous interface: 0
Feature concentration: 19
Unstable dependency: 25
Scattered functionality: 0
Dense structure: 1

==========
Design smells:
==========

v2.8.10

Imperative abstraction: 0
Multifaceted abstraction: 2
Unnecessary abstraction: 11
Unutilized abstraction: 156
Feature envy: 28
Deficient encapsulation: 84
Unexploited encapsulation: 1
Broken modularization: 5
Cyclically-dependent modularization: 62
Hub-like modularization: 2
Insufficient modularization: 67
Broken hierarchy: 90
Cyclic hierarchy: 2
Deep hierarchy: 0
Missing hierarchy: 1
Multipath hierarchy: 2
Rebellious hierarchy: 5
Wide hierarchy: 2

v2.8.9

Imperative abstraction: 0
Multifaceted abstraction: 2
Unnecessary abstraction: 11
Unutilized abstraction: 154
Feature envy: 28
Deficient encapsulation: 84
Unexploited encapsulation: 1
Broken modularization: 5
Cyclically-dependent modularization: 62
Hub-like modularization: 2
Insufficient modularization: 67
Broken hierarchy: 90
Cyclic hierarchy: 2
Deep hierarchy: 0
Missing hierarchy: 1
Multipath hierarchy: 2
Rebellious hierarchy: 4
Wide hierarchy: 2

v2.8.4

Imperative abstraction: 0
Multifaceted abstraction: 2
Unnecessary abstraction: 11
Unutilized abstraction: 158
Feature envy: 29
Deficient encapsulation: 82
Unexploited encapsulation: 1
Broken modularization: 4
Cyclically-dependent modularization: 55
Hub-like modularization: 2
Insufficient modularization: 65
Broken hierarchy: 90
Cyclic hierarchy: 2
Deep hierarchy: 0
Missing hierarchy: 1
Multipath hierarchy: 2
Rebellious hierarchy: 4
Wide hierarchy: 2

v2.8.3

Imperative abstraction: 0
Multifaceted abstraction: 2
Unnecessary abstraction: 11

Unutilized abstraction: 154
Feature envy: 29
Deficient encapsulation: 78
Unexploited encapsulation: 1
Broken modularization: 4
Cyclically-dependent modularization: 55
Hub-like modularization: 2
Insufficient modularization: 63
Broken hierarchy: 90
Cyclic hierarchy: 2
Deep hierarchy: 0
Missing hierarchy: 1
Multipath hierarchy: 2
Rebellious hierarchy: 4
Wide hierarchy: 2

==========
Implementation smells:
==========

v2.8.10

Abstract function call from constructor: 2
Complex conditional: 62
Complex method: 96
Empty catch clause: 20
Long identifier: 37
Long method: 17
Long parameter list: 109
Long statement: 824
Magic number: 1806
Missing default: 36

v2.8.9

Abstract function call from constructor: 2
Complex conditional: 63
Complex method: 96
Empty catch clause: 20
Long identifier: 37
Long method: 17
Long parameter list: 109
Long statement: 818
Magic number: 1793
Missing default: 36

v2.8.4

Abstract function call from constructor: 2
Complex conditional: 54
Complex method: 86
Empty catch clause: 16
Long identifier: 37
Long method: 15
Long parameter list: 104
Long statement: 760
Magic number: 1744
Missing default: 34

v2.8.3

Abstract function call from constructor: 2
Complex conditional: 52
Complex method: 86
Empty catch clause: 16
Long identifier: 34
Long method: 14
Long parameter list: 101
Long statement: 749
Magic number: 1717
Missing default: 34

==========
Analysis of the evolution of the design flaws as the software size evolves and point out which of the design qualities are in deficit:
==========

As the total design flaws increase from the first analyzed version to the last, there is also an increase in the total design flaw sum. In addition to this, the quality deficit index has also gone up as one climbs up to the latest version. The first version analyzed which is V2.8.3 has a quality deficit index of 6.9, V2.8.4 with 7, V2.8.9 with 6.8 and the latest V2.8.10 with 7.1.

This doesn't necessarily mean that the code is getting worse since this is based on the total lines of code. This may be that there's additional functionality added to the software. We can see that the more complex the software, there has been a decrease in the encapsulation deficit as well as cohesion deficit. The coupling deficit and inheritance deficit has increased as the more complex the code has become. This means that, the more complex the software becomes, there is more encapsulation and the code is more robust. The more cohesive the software is, the better. This is because

more cohesion means that the software is more focused on what it's doing. In terms of coupling, there has been an increase in coupling as the codebase gets bigger and bigger, depending on the ratio, this could be a good or bad thing. There has been an increase in inheritance deficit which means that there is a decrease in the total inheritance of the source code. Inheritance is facilitates reusability of the code, so based on the data, there hasn't been any improvement as the versions go up.

The data spit out by DesigniteJava also showed a similar phenomenon. As the complexity of the methods increase, the higher the design smells and architecture smells in general. A notable difference however is that Designite found a lot more Cyclic dependencies as well as Unstable dependencies compared to InFusion. But this data also rides with the findings of InFusion. Overall, both InFusion and Designite had similar results. The higher the lines of code, showed more complexity of each version and therefore, showed a lot more design flaws.

Links for the 4 versions:

https://github.com/bardsoftware/ganttproject/releases
https://github.com/bardsoftware/ganttproject/releases?after=ganttproject-2.8.9