Quality Assessment of Test Suites of Architectural Smelly Components

Manuel De Stefano, *SeSa Lab, University of Salerno,* Armando Conte, *University Of Salerno,*
Grazia Varone, *University Of Salerno,* and Fabio Palomba, *SeSa Lab, University of Salerno*

Abstract—The goal of this study is to perform an historical analysis of the test-suites related to components affected by architectural smells in open-source systems, with the purpose of assessing whether the quality of these test suites decreases when architectural smells are introduced. Nowadays project can contain several type of smells, in particular we focused on architectural and test smells, that can negatively impact on different software qualities. Several Architectural and test smells have been considered for our purpose, in particular for the Architectural we considered 18 type of smells: 6 at package level and the other 12 at class level. For test smells we considered 7 type of smells. We have analyzed a list of projects selected on GitHub by specific criteria, in order to detect all the chosen Architectural and test smells. We collected all these data about smells with the goal to detect if an increase of architectural smells leads to an increase of test smells and in particular the presence of which architectural smell lead to which test smell. To compute this, we used Association Rule method, showing that there are test smells like *Assertion Roulette* and *Eager Test* that more than other test smells can be dependent by the presence of an architectural smell.

Index Terms—Architectural Smells, Code quality, Code smells, Correlation, Inter-smell relationships.

1 Introduction

Architectural smells are one of the greatest sources of technical debt. An important task is to identify them and manage them, to prevent and avoid technical debt. The Architectural Smells (ASs) could be seen as some code smells but at the architectural level. These represent the violation of design principles or the decisions that impact the quality of internal software, with an increase of evolution and maintenance costs, but in particular, they impact the definition of test cases, because with some Architectural Smell in the code could increase the difficulty to test the code and some Test Smells can be introduced.

Hence, this project aims to find a relationship between the Architectural Smells and the Test Smells and in particular to observe if an increase of ASs will correspond to an increase of Test Smells.

To investigate our goal, we identified before a list of projects on GitHub to mine and observe the Architectural Smells and Test Smells vary over time, between different versions. Then we selected two tools, one that allows the detection of Architectural Smells and one that allows the detection of Test Smells. According to our aim, we collected the following data: (i) the list of projects to mine on GitHub, (ii) the list of the Architectural Smells detected by the tool for every version of each project, (iii) the list of the Test Smells detected by the tool for every version of each project.

Hence, in this project, we aim to answer the following questions:

- Q1: With an increase of Architectural Smells will we have an increase of Test Smells too?
- Q2: If we have one Architectural Smell what Test Smell will it imply?

The answers obtained for the previous questions can be useful for several reasons, in particular to software developers/maintainers to be able to know if they have a particular Architectural Smell what Test Smell could imply.

The document is organized through the following sections: in Section II we introduce the empirical study design that has been done by describing (i) how the projects have been selected, (ii) how the Architectural Smells have been identified (iii) how the Test Smells have been identified (iii) how the relationship between Architectural Smells and Test Smells has been found. Section III contains the results obtained. Finally, in Section IV we conclude our work and describe some future developments that can be done.

2 Empirical Study Design

The *goal* of the study is to perform an historical analysis of the test-suites related to components affected by architectural smells in open-source systems, with the *purpose* of assessing whether the quality of these test suites decreases when architectural smells are introduced. Moreover, the study aims to asses how the fault proneness of the considered components varies when these smells occur.

2.1 Context selection

The context of our study is made up of architectural smells and software systems. Among the currently known architectural smells , we considered the following package level Architectural Smells:

- **God Component:** the component contains a high number of classes.

- **Cyclic Dependency:** this component participates in a cyclic dependency.
- **Unstable Dependency:** this component depends on other components that are less stable than itself.
- **Feature Concentration:** the component realizes more than one architectural concern/feature.
- **Dense Structure:** all the analyzed components exhibit excessive and dense dependencies among themselves.
- **Ambiguous Interface:** the component provides only a single general entry-point via a class.

Moreover, Hub-Like Dependency and Cyclic Dependency are well-known smells and object of a great number of studies [1]. However, for the opposite reason, we chose to focus on class level Architectural Smells, since, as explained by [1], they have never been studied, and so we focused on:

- **Deficient Encapsulation:** This smell occurs when the declared accessibility of one or more members of abstraction is more permissive than required.
- **Unutilized Abstraction:** This smell arises when an abstraction is left unused (either not directly used or not reachable).
- **Feature Envy:** This smell occurs when a method seems more interested in a class other than the one it is in.
- **Broken Hierarchy:** This smell arises when a supertype and its sub-type conceptually do not share an ”IS-A” relationship resulting in broken substitutability.
- **Broken Modularization:** This smell arises when data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions.
- **Insufficient Modularization:** This smell arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both.
- **Wide Hierarchy:** This smell arises when an inheritance hierarchy is ”too” wide indicating that intermediate types may be missing.
- **Unnecessary Abstraction:** This smell occurs when an abstraction that is not needed (and thus could have been avoided) gets introduced in a software design.
- **Multifaceted Abstraction:** This smell arises when an abstraction has more than one responsibility assigned to it.
- **Cyclically-dependent Modularization:** This smell arises when two or more abstractions depend on each other directly or indirectly.
- **Cyclic Hierarchy:** This smell arises when a supertype in a hierarchy depends on any of its sub-types.
- **Rebellious Hierarchy:** This smell arises when a subtype rejects the methods provided by its supertype(s).

We chose these because they all occur at the class level, so we could conduct our study at the same level of granularity. To compute these Architectural Smells we have chosen to use a tool called Designite [2], which allows the computation of all the smells previously described for the Java projects.

Regarding Test Smells we focused on static analysis to detect these and in particular we considered the following:

- **It1 (Ignored Test):** A test method or class that contains the `@Ignore` annotation.
- **Gf1 (General Fixture):** Not all fields instantiated within the `setUp()` method of a test class are utilized by all test methods in the same test class.
- **Ro1 (Resource Optimism):** A test method utilizes an instance of a `File` class without calling the `exists()`, `isFile()` or `notExists()` methods of the object.
- **Ar1 (Assertion Roulette):** A test method contains more than one assertion statement without an explanation/message (parameter in the assertion method).
- **Et1 (Eager Test):** A test method contains multiple calls to multiple production methods.
- **Mg1 (Mystery Guest):** A test method containing object instances of files and databases classes.
- **Se1 (Sensitive Equality):** A test method invokes the `toString()` method of an object.

2.2 Data analysis

For the computation of these metrics has been chosen the tool VITRuM [3], this tool was only a tool for IntelliJ Plugin, but it was adapted to be used by Command Line Interface too, and for this reason, we used this tool for static analysis of test cases.

The first step was the selection of GitHub projects and to chose what project mine we used these criteria:

- Only Java project.
- Minimum release number: 10.
- Minimum star number: 10.000.
- Exclude fork.

With these criteria, we selected 90 projects, and on these projects, we used the framework Repodriller that allows the mining of the repository [4]. Using this framework we mined all these repository releases and for every release, we launched Designite, to detect Architectural Smells, and VITRuM, to detect Test Smells, to collect the information about all these smells. All the results returned for every release were merged in a unique CSV file with some additional columns, in particular, the columns are:

- **HashTag:** the name of the current release.
- **HashCommit:** the hash of the corresponding commit.
- **Date:** the date of the commit.

We added these three columns for both Architectural Smells and Test Smells CSV files.

NameTag	HashCommit	Date
androidannotations-2.2-RC1	c4245b6ffce3c5eae94c9b9e6c66a422af3c5bd8	Sat Nov 26 18:34:10 CET 2011
androidannotations-2.2-RC1	c4245b6ffce3c5eae94c9b9e6c66a422af3c5bd8	Sat Nov 26 18:34:10 CET 2011
androidannotations-2.2-RC1	c4245b6ffce3c5eae94c9b9e6c66a422af3c5bd8	Sat Nov 26 18:34:10 CET 2011

Fig. 1. Example of the three added column

Unfortunately, during the mining, there were some problems with the computation of Designite for some projects,

because it could not detect smells and for this reason, they were put aside. There were some problems with VITRuM too and in particular, we couldn't analyze some projects because they didn't have test cases and so there was nothing to analyze or some projects didn't have the correct structure `src/test` that VITRuM finds with the aim of analyzing the projects. The problem of the structure was present when the project was organized in modules, and to resolve this we created a recursive function that analyzed these modules and runs VITRuM on single modules, after that the results were merged all in one. In some other cases, some projects did not have the test folder called `test`, for example, they had `testRun` or similar, and so VITRuM couldn't find it and consequently couldn't analyze the test cases of these projects. For these reasons, the number of projects passed from 90 to 40, but fortunately, it was not a small number, especially because we selected only big projects with a minimum of 10 releases, and it allowed us to conduct the analysis of the resultant data.

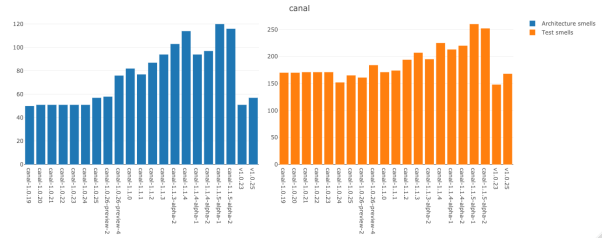

Fig. 3. Project: Canal

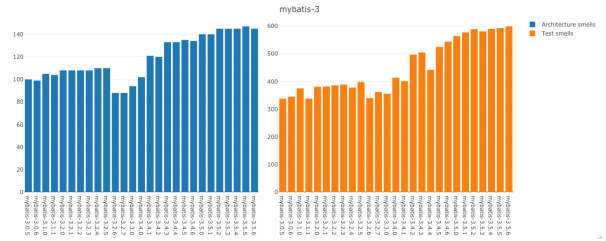

Fig. 4. Project: Mybatis-3

3 Analysis of Results

After collecting all the data of the Architectural and Test smells we analyzed them. This section discusses the obtained results with the goal of answering the two formulated questions. In particular, we will present the findings for Q1 in Section 3.1 and Q2 in Section 3.2 to facilitate discussion of the results and avoid redundancies.

3.1 Q1: Correlated increase of Architectural and Test smells

For every project that we mined and gathered data we compared this data and created a pair of plots, in particular:

- The first represents the number of Architectural Smells for the entire project.
- The second represents the number of Test Smells for the entire project.

The goal of this phase was to observe visually, through a graphical representation, that with an increase of the number of Architectural Smells for the project there was an increase of Test Smells too.

In the images below you can see this parallel increment between the two plots of the same project.

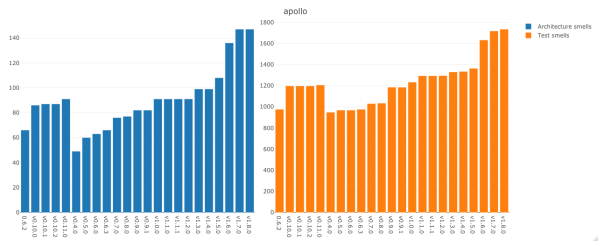

Fig. 2. Project: Apollo

For reason of space we reported only a few of the plots of the projects analyzed, but all the other plots are available on the Github repository [5].

As you can observe from the previous plots when there is an increase of Architectural smells (blue plots) there is an increase of Test smells (orange plots) too, and it is also interesting to observe that when the Architectural smells decrease there is a decrease of the Test smells too.

These plots are related to the versions of the projects, so every bar of these represent the number of smells contained in the relative version, while the Architectural smells considered are package-level.

Once computed these, we decided to compute the co-occurrence of the smells[1], in particular we calculated all the co-occurrence between:

- *Architectural Smells - Architectural Smells*;
- *Test Smells - Test Smells*;
- *Architectural Smells - Test Smells*;

As you can see in the Table 1 the Architectural Smells considered are at the class level because we compared them with Test Smells that are at a class level too. In the Table 2 there are the co-occurrence of Test Smells, and in the table 3 there are the co-occurrence between the Architectural Smells and Test Smells at class level. The tables in the examples refers to the specific project *"Apollo"* [9]. In the GitHub repository are available the Python scripts to compute it and the result of the computation for all the projects [6].

The Table 1 shows clearly that there is the 96% of probability to find both *Unnecessary Abstraction* and *Unutilized Abstraction*, for example. According to test smells, the table 2 shows that there is the 93% of probability to find *ar1* and *et1* together. Finally, in the table 3, you can see that when there is the smell *Wide Hierarchy* with the 79% of probability there will be a test smell *ar1*.

1. The counting of paired data within a collection.

TABLE 1
Co-occurrences Architectural Smell - Architectural Smell in *Apollo* project.

	Unnecessary Abstraction	Unutilized Abstraction	Deficient Encapsulation	Feature Envy	Cyclically-dependent Modularization	Insufficient Modularization	Hub-like Modularization	Broken Hierarchy	Broken Modularization	Wide Hierarchy	Multifaceted Abstraction	Multipath Hierarchy
Unnecessary Abstraction	1.0	0.96	0.13	0.0	0.0	0.23	0.0	0.27	0.0	0.06	0.02	0.0
Unutilized Abstraction	0.13	1.0	0.05	0.12	0.01	0.07	0.0	0.2	0.01	0.01	0.0	0.0
Deficient Encapsulation	0.2	0.59	1.0	0.3	0.1	0.11	0.0	0.14	0.11	0.05	0.01	0.0
Feature Envy	0.0	0.6	0.14	1.0	0.1	0.18	0.02	0.21	0.0	0.0	0.01	0.0
Cyclically-dependent Modularization	0.0	0.35	0.33	0.72	1.0	0.61	0.17	0.18	0.0	0.0	0.01	0.0
Insufficient Modularization	0.33	0.74	0.11	0.37	0.18	1.0	0.05	0.45	0.0	0.09	0.05	0.0
Hub-like Modularization	0.0	1.0	0.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0
Broken Hierarchy	0.11	0.59	0.03	0.12	0.01	0.12	0.0	1.0	0.01	0.04	0.01	0.01
Broken Modularization	0.01	0.23	0.31	0.0	0.0	0.01	0.0	0.08	1.0	0.01	0.01	0.0
Wide Hierarchy	0.63	1.0	0.37	0.0	0.0	0.63	0.0	1.0	0.02	1.0	0.26	0.0
Multifaceted Abstraction	0.36	0.49	0.14	0.14	0.02	0.49	0.0	0.37	0.03	0.36	1.0	0.0
Multipath Hierarchy	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0

TABLE 2
Co-occurrences Test Smell - Test Smell in *Apollo* project.

	ar1	et1	it1	gf1	mg1	ro1	se1
ar1	1.0	0.93	0.06	0.43	0.06	0.04	0.01
et1	0.92	1.0	0.06	0.44	0.07	0.04	0.02
it1	0.98	1.0	1.0	0.63	0.0	0.0	0.03
gf1	0.98	1.0	0.08	1.0	0.12	0.06	0.02
mg1	0.94	0.98	0.0	0.81	1.0	0.6	0.0
ro1	0.9	0.97	0.0	0.68	1.0	1.0	0.0
se1	0.37	1.0	0.07	0.3	0.0	0.0	1.0

TABLE 3
Co-occurrences Architectural Smell - Test Smell in *Apollo* project.

	ar1	et1	it1	gf1	se1	mg1	ro1
Deficient Encapsulation	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Unutilized Abstraction	0.01	0.0	0.0	0.0	0.0	0.0	0.0
Feature Envy	0.19	0.19	0.01	0.0	0.03	0.03	0.01
Broken Hierarchy	0.2	0.18	0.04	0.11	0.04	0.0	0.0
Broken Modularization	0.32	0.32	0.13	0.13	0.0	0.0	0.0
Insufficient Modularization	0.14	0.14	0.0	0.06	0.04	0.0	0.0
Wide Hierarchy	0.79	0.79	0.0	0.0	0.0	0.0	0.0
Unnecessary Abstraction	0.09	0.1	0.0	0.07	0.02	0.01	0.02
Multifaceted Abstraction	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Cyclically-dependent Modularization	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Cyclic Hierarchy	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Rebellious Hierarchy	0.0	0.0	0.0	0.0	0.0	0.0	0.0

3.2 Q2: Find association between smells through Association rule

After that, our goal was to find an association between different objects in a set, find frequent patterns in any information repository. In particular, our purpose was to generate a set of rules called Association Rules, in form **if this then that**.

Also in this case we considered the three combination of (i) *Architectural Smells - Architectural Smells*, (ii) *Test Smells - Test Smells*, (iii) *Architectural Smells - Test Smells*, where for Architectural Smells we refer to class level. What we have done here was to create a unique dataset which contains all the Architectural Smells for all the projects and a unique dataset that contains all the Test Smells for all the projects. After that, we did the inner join between them on the attribute className (in fact for this reason we considered Architectural Smells at class level). The result of this operation was a big dataset that contains for each class several rows with all the smells for all the versions of all the projects. For simplicity in reading, we had added an additional column that contained the smells of that row separated by commas.

Before applying Association Rule mining, we needed to convert data frame into transaction data so that all smells that appeared together in one class were put in one row. The smells were grouped using NameClass. We did this in R with:

```
library(plyr)

transactionData <- ddply(df_join,c("ClassName"),
          function(df_join)paste(df_join$Smells,
                                 collapse = ","))
```

These data then were saved in a CSV file and then was loaded into an object of the transaction class. This was done by using the R function `read.transactions` with the basket format and converted it into an object of the transaction class. With the summary of the transactions we obtained:

```
## transactions as itemMatrix in sparse format with
##  1269 rows (elements/itemsets/transactions) and
##  25 columns (items) and a density of 0.1374626
##
## most frequent items:
##                          ar1                          et1
##                         1023                          998
##     Deficient Encapsulation Insufficient Modularization
##                          369                          336
##      Unutilized Abstraction                      (Other)
##                          332                         1303
##
## element (itemset/transaction) length distribution:
## sizes
##   1   2   3   4   5   6   7   8   9
##  61 183 454 372 135  47  11   4   2
##
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   3.000   3.000   3.437   4.000   9.000
##
## includes extended item information - examples:
##                  labels
## 1                   ar1
## 2      Broken Hierarchy
## 3 Broken Modularization
```

Fig. 5. Summary of transaction results

In particular:

- There are 1269 transactions (rows) and 25 items (columns). Note that 25 are the smells involved in the dataset and 1269 transactions are collections of these items.
- Density tells the percentage of non-zero cells in a sparse matrix. You can say it as the total number of smells that are detected divided by a possible number of smells in that matrix. You can calculate how many items were purchased by using density: $1269 * 25 * 0.1374626 = 4361.0$.

Then we computed the item frequency plots for both absolute and relative because if absolute it will plot numeric frequencies of each item independently, if relative it will plot

how many times these items have appeared as compared to others.

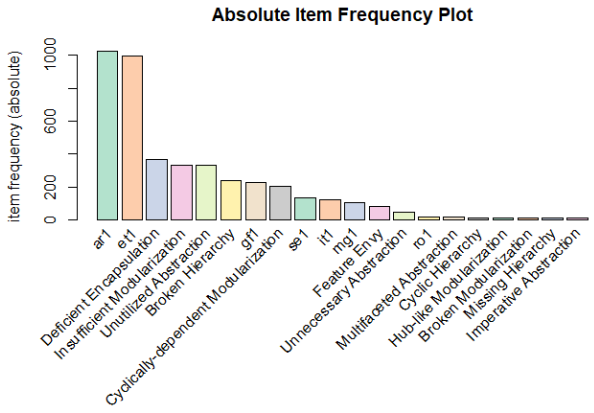

Fig. 6. Absolute Item Frequency Plot

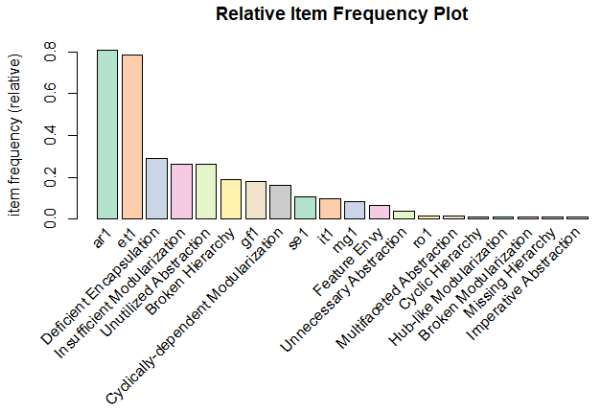

Fig. 7. Relative Item Frequency Plot

The next step was to mine the rules using the **APRIORI** algorithm. We used the function `apriori()` from package `arules`. The apriori takes the transaction data as the transaction object on which mining is to be applied. The parameter that we used for the support was 0.001, and for the confidence was 0.8 (because several works have chosen these thresholds like [8]), then we specified a maximum of 2 items (maxlen), minimum of 2 items (minlen). With the appearance, we specified the LHS (IF part) with the array of Architectural Smells and RHS (THEN part) with the Test Smells.

```
design_smells = c('Deficient Encapsulation', 'Unutilized
    Abstraction', 'Feature Envy',  'Broken Hierarchy', '
    Broken Modularization', 'Insufficient Modularization',
    'Wide Hierarchy', 'Unnecessary Abstraction', '
    Multifaceted Abstraction', 'Cyclically-dependent
    Modularization','Cyclic Hierarchy', 'Rebellious
    Hierarchy')

test_smells = c("ar1", "et1", "it1",  "gf1", "se1", "mg1",
    "ro1")

association.rules <-
apriori(tr, parameter = list(supp=0.001,
           conf=0.8, minlen=2, maxlen=2),
   appearance = list(lhs=design_smells, rhs=test_smells))
```

From this computation with these parameters we obtained 9 rules.

```
##     lhs                                      rhs    support    confidence
## [1] {Cyclic Hierarchy}                    => {et1} 0.01024429 0.8125000
## [2] {Multifaceted Abstraction}            => {ar1} 0.01497242 1.0000000
## [3] {Feature Envy}                        => {et1} 0.06067770 0.8953488
## [4] {Feature Envy}                        => {ar1} 0.05910165 0.8720930
## [5] {Cyclically-dependent Modularization} => {et1} 0.14657210 0.9162562
## [6] {Insufficient Modularization}         => {et1} 0.25374310 0.9583333
## [7] {Insufficient Modularization}         => {ar1} 0.22616233 0.8541667
## [8] {Deficient Encapsulation}             => {et1} 0.24586288 0.8455285
## [9] {Deficient Encapsulation}             => {ar1} 0.25137904 0.8644986
```

Fig. 8. Rule Architectural Smells - Test Smells

Using the above output, you can make analyses such as:

- Classes which have design smell *'Cyclic Hierarchy'* will have the test smell *'et1'* with support of 0.01 and confidence of 0.81.
- Classes which have design smell *'Multifaceted Abstraction'* will have the test smell *'ar1'* with support of 0.0149 and confidence of 1.
- Classes which have design smell *'Feature Envy'* will have the test smell *'et1'* with support of 0.06 and confidence of 0.89.
- Classes which have design smell *'Feature Envy'* will have the test smell *'ar1'* with support of 0.059 and confidence of 0.87.

We did the same thing with both LHS (left part) and RHS (right part) imposed with Architectural Smells, but here we setted the parameter maxlen of the apriori function equals to 4 because with maxlen=2 or maxlen=3 there were less than 2 rules. From this computation we obtained the following 7 rules:

```
##     lhs                                      rhs                          support confidence
## [1] {Broken Modularization,
##      Unnecessary Abstraction}            => {Unutilized Abstraction}     0.003940110  1.0
## [2] {Feature Envy,
##      Unnecessary Abstraction}            => {Unutilized Abstraction}     0.002364066  1.0
## [3] {Broken Modularization,
##      Deficient Encapsulation,
##      Unnecessary Abstraction}            => {Unutilized Abstraction}     0.002364066  1.0
## [4] {Broken Modularization,
##      Deficient Encapsulation,
##      Unutilized Abstraction}             => {Unnecessary Abstraction}    0.002364066  1.0
## [5] {Cyclic Hierarchy,
##      Cyclically-dependent Modularization,
##      Deficient Encapsulation}            => {Insufficient Modularization} 0.001576044  1.0
## [6] {Cyclically-dependent Modularization,
##      Deficient Encapsulation,
##      Feature Envy}                       => {Insufficient Modularization} 0.003152088  0.8
## [7] {Broken Hierarchy,
##      Cyclically-dependent Modularization,
##      Deficient Encapsulation}            => {Insufficient Modularization} 0.001576044  1.0
```

Fig. 9. Rule Architectural Smells - Architectural Smells

Using the above output, you can make analyses such as:

- Classes which have design smells *'Broken Modularization'* and *'Unnecessary Abstraction'* will have the design smell *'Unutilized Abstraction'* with support of 0.0039 and confidence of 1.
- Classes which have design smell *'Feature Envy'* and 'Unnecessary Abstraction' will have the design smell *'Unutilized Abstraction'* with support of 0.0023 and confidence of 1.

Finally, we did the same computation to Test Smells to and we called the apriori function with both LHS (left part) and RHS (right part) imposed with Test Smells but here we imposed the maxlen and minlen both equals to 2. From this computation, we obtained 13 rules and we filtered the first 10:

```
##      lhs      rhs    support    confidence
## [1]  {ro1} => {mg1} 0.01260835 0.8000000
## [2]  {ro1} => {et1} 0.01418440 0.9000000
## [3]  {ro1} => {ar1} 0.01418440 0.9000000
## [4]  {mg1} => {et1} 0.07407407 0.8867925
## [5]  {mg1} => {ar1} 0.07171001 0.8584906
## [6]  {it1} => {et1} 0.08983452 0.9120000
## [7]  {it1} => {ar1} 0.08747045 0.8880000
## [8]  {se1} => {et1} 0.09771474 0.9185185
## [9]  {se1} => {ar1} 0.09613869 0.9037037
## [10] {gf1} => {et1} 0.15602837 0.8608696
```

Fig. 10. Rule Test Smells - Test Smells

Using the above output, you can make analyses such as:

- Classes which have test smell *'ro1'* will have the test smell *'mg1'* with support of 0.0126 and confidence of 0.8.
- Classes which have test smell *'ro1'* will have the test smell *'et1'* with support of 0.0141 and confidence of 0.9.

3.2.1 Visualize Association Rules

Since there will be hundreds or thousands of rules generated based on data, there are a couple of ways to visualize these association rules.

One of the visualization that we computed is the Individual Rule Representation, this representation is also called as Parallel Coordinates Plot.

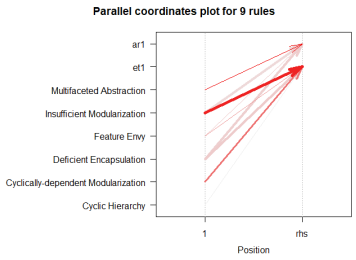

Fig. 11. Parallel Coordinate Architectural Smells-Test Smell

For example, Figure 13 shows that when the class has the test smell *'ro1'*, with high probability it will also have the test smell *'mg1'*. All these plots have been created throw R, the scripts and a report of these results are all reported on GitHub [7].

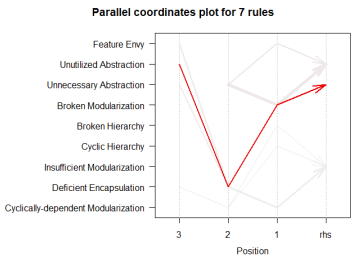

Fig. 12. Parallel Coordinate Architectural Smells

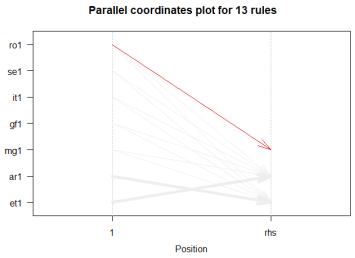

Fig. 13. Parallel Coordinate Test Smells

4 Conclusion

BOTH Architectural and Test smells, as smells are symptoms of poor design or implementation choices. What we wanted to present in this project was a relationship between the Architectural smells and Test smells, in particular, in the first moment we computed the co-occurrences among them for all the smells of the projects that we selected and mined on GitHub, by providing evidence on what are the smells that co-occur more frequently not only among Architectural-Test but also between Architectural-Architectural and Test-Test smells.

After that our focus was on the Association Rule because our principal purpose was to show that the introduction of an Architectural smell increase the difficulty to write the relative test case, because of the complexity of the structure of the code to test, and therefore the developer while testing these smelly components will introduce some Test smells. At this point, we computed the Association Rule, through the Apriori function, on all the smells of the projects that we collected, to discover what Architectural smell lead to what Test smell and what is its support and its confidence. What we concluded was that various Architectural smell lead more frequency to the Test smell *"ar"* and *"et"*. In particular we have observed that the number of ET tests smell increases when there are arch smells because the system architecture becomes more complex and consequently more difficult to test, for this reason when you go to test you have the risk that the tests do not have the focus on a single target, consequently they can have more asserts and this leads to smell ET. We computed also the Association rule between the Architectural-Architectural and Test-Test smells.

These findings represent the main results of our project, but also some other researches could be done to discover eventually other rules of Association Rule by considering new Architectural smells and Test smells in addition to those already considered.

REFERENCES

[1] H. Mumtaz and P. Singh and K. Blincoe, *A systematic mapping study on architectural smells detection*, University of Auckland, New Zealand: The Journal of Systems & Software, 2021.

[2] T. Sharma and P. Mishra and R. Tiwari, *Designite - A Software Design Quality Assessment Tool*, 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities, 2016.

[3] F. Pecorelli and G. Di Lillo and F. Palomba and A. De Lucia, *VIT-RuM: A Plug-In for the Visualization of Test-Related Metrics*, University of Salerno, Italy: International Conference on Advanced Visual Interfaces, 2020.

[4] *RepoDriller*, https://github.com/mauricioaniche/repodriller, accessed: 2021-06-30.

[5] *ProgettoSwD*, https://github.com/mdestefano/progetto-swd/tree/main/scripts/datasets/graficiR, accessed: 2021-06-30.

[6] *ProgettoSwD*, https://github.com/mdestefano/progetto-swd/tree/main/scripts/co-occurrence, accessed: 2021-06-30.

[7] *ProgettoSwD*, https://https://github.com/mdestefano/progetto-swd/tree/main/scripts/association-rule, accessed: 2021-06-30.

[8] *A Gentle Introduction on Market Basket Analysis Association Rules*, https://towardsdatascience.com/a-gentle-introduction-on-market-basket-analysis-association-rules-fa4b986a40ce, accessed: 2021-07-01.

[9] *Apollo - A reliable configuration management system*, https://github.com/ctripcorp/apollo, accessed: 2021-07-02.