Architecture Analysis Report

2020-05-21

ArchDia INC

Contents

[1. Introduction 2](#_Toc535934758)

[2. Maintainability Scores 2](#_Toc535934759)

[2.1 Maintainability Metrics Introduction 2](#_Toc535934760)

[2.2 Project Scores 3](#_Toc535934761)

[3. Architectural Anti-Pattern Instances and Their Severity Rankings 3](#_Toc535934762)

[3.1 Architectural Anti-pattern Introduction 3](#_Toc535934763)

[3.2 Architectural Anti-patterns Detected 4](#_Toc535934764)

[4 Architectural Root Analysis and Architectural Debt Calculation 4](#_Toc535934765)

[4.1 Architectural Roots Introduction 4](#_Toc535934766)

[4.2 Architectural Roots Detected 4](#_Toc535934767)

[5. Return on Investment Analysis 5](#_Toc535934768)

[6. Summary 5](#_Toc535934769)

[References 6](#_Toc535934770)

# Introduction

This report summaries DV8 analysis results, including

1. Maintainability/evolvability scores of source code structure, measured using state-of-the-art metrics suites, as reported in [1].
2. Architecture anti-pattern analysis. DV8 identifies 6 types of architecture anti-patterns as defined in [2]. Each instance is represented as a Design Structure Matrix (DSM) files [1][2] that can be opened using DV8 Explorer.
3. Architecture root analysis. We define Architecture Roots as a set of design spaces that cover most error-prone and/or change-prone files, as introduced in [3].
4. Return on Investment (ROI) analysis. Considering each root as an architectural debt, DV8 also provides a return on investment analysis that quantifies the potential benefit of refactoring problematic areas of the architecture, as reported in [4].

DV8 accepts the following input data:

* The dependency data extracted from a single snapshot of the project. This snapshot contains 1385 source files.
* The project's revision history. These records cover the evolution history of the project for the period 2019-04-07 to 2020-04-15.
* The issue records that contains 685 issues.

In the rest of the report, we refer to the project as *Proj\_All*.

# Maintainability Scores

## 2.1 Maintainability Metrics Introduction

In this section, we introduce the concepts of the metrics that we employ, and our industrial benchmark data, formed by over 200 industrial and open source projects.

*Propagation Cost (PC).* MacCormack et al.’s Propagation Cost metric—calculated based on a DSM representation of a system’s dependencies —aims to measure how tightly coupled a system is. Given a DSM of a project’s dependencies, they first calculate its transitive closure to add indirect dependencies to the DSM until no more can be added. Given the final DSM with all direct and indirect dependencies, PC is calculated as the number of non-empty cells divided by the total number of cells.

*Decoupling Level (DL).* Decoupling Level measures how well an architecture is decoupled into modules. To calculate DL DV8 first clusters a DSM into a design rule hierarchy (DRH). The more files that a given module influences in lower layers, the lower its DL. In addition, the larger a module, the more likely it will influence more files in the lower layers, and hence the lower its DL. The more independent modules are there, the higher the DL.

Thus, for an architecture, a higher DL is better, and a lower PC is better. Table 1 below shows the average levels of DL and PC for a set of 221 open source projects with more than 100 files, and 21 commercial projects.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Stats** | **Open Source** | | **Commercial** | | **All Projects** | |
|  | *DL* | *PC* | *DL* | *PC* | *DL* | *PC* |
| Average | **0.59** | **0.22** | **0.54** | **0.21** | **0.59** | **0.22** |
| Median | 0.59 | 0.19 | 0.56 | 0.2 | 0.59 | 0.23 |
| Max | 0.92 | 0.78 | 0.93 | 0.5 | 0.93 | 0.78 |
| Min | 0.14 | 0.02 | 0.15 | 0.02 | 0.15 | 0.02 |
| 20th Pt | 0.43 | 0.35 | 0.36 | 0.35 | 0.42 | 0.35 |
| 40th Pt | 0.54 | 0.22 | 0.46 | 0.24 | 0.54 | 0.22 |
| 60th Pt | 0.67 | 0.15 | 0.59 | 0.17 | 0.66 | 0.15 |
| 80th Pt | 0.76 | 0.07 | 0.65 | 0.06 | 0.76 | 0.07 |

Table 1: Industrial Benchmark Data

## 2.2 Project Scores

Following are the architectural scores for the system, measured using Decoupling Level (DL), and Propagation Cost (PC), as introduced in Section 2.1. Again, a higher DL is better, and a lower PC is better. If your system has a DL of 0.5 or less, you should consider refactoring.

This system has 1385 files, of which 85 are isolated from the rest.

- Considering all the files, the Decoupling Level is 71.75%, and the Propagation Cost is 14.68%.

- Considering only the connected files (excluding isolated ones), the Decoupling Level is 61.63%, and the Propagation Cost is 16.66%.

# Architectural Anti-Pattern Instances and Their Severity Rankings

## 3.1 Architectural Anti-pattern Introduction

DV8 currently detects the following types of architecture anti-patterns:

* *Clique:* a group files that are interconnected, forming a strongly connected “component” but not belonging to a single module.
* *Package cycles:* typically the package structure of a software system should form a hierarchical structure. A cycle among packages is therefore considered to be harmful.
* *Improper inheritance:* we consider an inheritance hierarchy to be problematic if it falls into one of the following cases: (1) a parent class depends on one or more of its children; (2) the client of the class hierarchy uses/calls both a parent and one or more of its children, thus violating the Liskov Substitution Principle.
* *Modularity violation*: properly designed modules—ones designed with information hiding in mind—should be able to change independently from each other. If two structurally independent modules in a DSM are shown to change together frequently in the revision history, it means that they are not truly independent from each other. We observe that in many of these cases, these modules have harmful implicit dependencies that should be removed. We call this flaw a modularity violation.
* *Crossing:* if a file has many dependents and depends on many other files than this file will appear to be at the center of a cross when visualized in a DSM. We call such a flaw a *Crossing*.
* *Unstable interface:* if a highly influential file is changed frequently with other files that directly or indirectly depend on it, then we call it an Unstable Interface. The default setting of DV8 is that: a file to be an unstable interface if it changes together with at least 10 other files 2 times or more. There parameters can be configured differently by the user.

These flaws have been shown to be strongly correlated to bugs, changes, and churn. That is, the more flaws a file is involved with, the more likely it will be buggy, change-prone, and costly to maintain [5].

## 3.2 Architectural Anti-patterns Detected

The following table summarizes the architectural anti-patterns detected in this project. This table also shows the maintenance costs incurred in the files within each type of anti-pattern including

* #Files (%): the number (percentage) of files participated in each anti-pattern
* #Bug Commits (%): the number (percentage) of bug commits incurred in files participated in each anti-pattern
* #Changes (%): the number (percentage) of changes incurred in files participated in each anti-pattern
* Bug Churn (%): the number (percentage) of bugs incurred in files participated in each anti-pattern
* Change Churn (%): the number (percentage) of changes incurred in files participated in each anti-pattern

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Architecture Flaw | Instances | #Files (%) | #Bug Commits (%) | Bug Churn (%) | #Changes (%) | Change Churn (%) |
| Crossing | 17 | 123 (8.9%) | - | - | 1,859 (40.7%) | 71,112 (46.7%) |
| UnstableInterface | 12 | 165 (11.9%) | - | - | 2,209 (48.3%) | 81,803 (53.7%) |
| Clique | 28 | 196 (14.2%) | - | - | 1,634 (35.7%) | 40,850 (26.8%) |
| ModularityViolation | 64 | 336 (24.3%) | - | - | 3,185 (69.7%) | 114,062 (74.8%) |
| UnhealthyInheritance | 126 | 438 (31.6%) | - | - | 2,547 (55.7%) | 97,340 (63.9%) |
| PackageCycle | 85 | 463 (33.4%) | - | - | 2,271 (49.7%) | 73,023 (47.9%) |
| Project Total |  | 1,385 | - | - | 4,572 | 152,434 |

Table 2: Architectural Flaws in Proj\_All

# 4 Architectural Root Analysis and Architectural Debt Calculation

## 4.1 Architectural Roots Introduction

*Architecture Roots* are the groups of files responsible for the most changes/bugs in the system. These architectural roots contain files (and their relations) that have the most impact on the maintainability of the system because error-proneness and change-proneness can propagate through their architectural relations. Research have shown that most error-prone files can be covered by just a few design spaces—Roots—and these roots are an output of DV8 root analysis.

## 4.2 Architectural Roots Detected

In this project, we obtained a set of Architectural Roots as summarized in the following table, that is, the groups of files are responsible for the most maintenance effort in the project. The meanings of each column of the tables in this section are as follows:

* DRSpace\_size: the number of files within the root
* Coverage (%): the proportion of bug-prone files covered by this DRSpace
* Cover\_up\_to (%): the proportion of bug-prone files covered by the top *n* DRSpaces

|  |  |  |  |
| --- | --- | --- | --- |
|  | DRSpace\_size | Coverage (%) | Cover\_up\_to (%) |

Table 3: Architectural Roots

Research has shown that the flawed architectural structures that propagate change-proneness or error-proneness to large numbers of a project’s files are a kind of *architectural debt.* If these flaws are not removed, extra maintenance costs will accumulate in the form of *penalty*, or *interest*, on the debt. [3][4]

# 5. Return on Investment Analysis

*Architecture Debt* is defined as the extra maintenance effort caused by the flawed relations *among Architectural Roots*. DV8 can quantify architectural debts as the additional number of changes and extra lines of code spent on the maintenance of the most prominent *Roots*.

In this project, we consider each of the Architectural Root as an architectural debt, and calculate its *penalty*, that is, the extra lines of code (LOC) spent, and the extra number of changes made. These debts, if paid down, can result in significant ROI (return-on-investment). The result is summarized in the following tables.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | #Files | Extra Bug Commits | Extra Bug LOC | Extra Changes | Extra Change LOC |

Table 4: Architectural Debt ROI

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Total #Files | Total Bug Commits | Total Bug LOC | Total Changes | Total Changed LOC |

Table 5: Architectural Debt Quantification

# 6. Summary

This report has summarized the outputs from the DV8 analysis.

We detected a number of architectural anti-pattern instances that have incurred significant maintenance costs. We have also detected a number of Architectural Roots. These anti-patterns and Roots are typically the foundation of much of a software project’s technical debt.

To reduce overall architecture debt, the problems within these architecture roots and anti-patterns should be considered as potential targets of refactoring.

# References

[1] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, Qiong Feng, “Decoupling level: a new metric for architectural maintenance complexity”. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE 2016, Page 499-510.

[2] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, “Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells”. In *Proceedings of* *12th Working IEEE/IFIP Conference on Software Architecture,* WICSA 2015: 51-60.

[3] Lu Xiao, Yuanfang Cai, Rick Kazman, “Design rule spaces: a new form of architecture insight”. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, Page 967-977.

[4] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, Andriy Shapochka, “A Case Study in Locating the Architectural Roots of Technical Debt”. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE 2015, Page 179-188.

[5] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, “Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells”, *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture* *(WICSA 2015*), May 2015.