Paper Reading Summary

王杰 2022年7月21日

---

1. Architecture Anti-patterns: Automatically Detectable Violations of Design Principles

Mo, R., Cai, Y., Kazman, R., Xiao, L., & Feng, Q. (2021). Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, *47*(5), 1008–1028. https://doi.org/10.1109/tse.2019.2910856

Mo是 Central China Normal University 华中师范大学，他的另外一篇论文就是ABB公司研究的那篇 实际上就是继承的工作

定义：

In this paper, we define a suite of **architecture anti-patterns** based on **Baldwin and Clark's design rule theory** [17] and widely accepted design principles, like SOLID principles(proposed by Robert Martin)

经过分析d大量industrial and open-source projects, 总结出6种经常出现的 Architecture Anti-Patterns, each **violating** 多于一种 design principles and/or design rule theory.

Baldwin and Clark's design rule theory：

a well-modularized system should have the following features:

- first, **design rules have to be stable**—neither error-prone nor changeprone.
- Second, if two modules are truly independent, then they should **only depend on design rules**, but not on each other.
- More importantly, independent modules should be able to be changed, or even replaced, without influencing each other, as long as the design rules themselves remain unchanged.

个人理解：设计规则理论可以视作一系列的围栏，作为某种code层面的尺度去隔离开不同的模块，围栏的种类和样式可以不同，但要满足上面三条原则

理论意义： **abstractions**, are also reflected in the Liskov substitution, Interface segregation, and Dependency inversion principles. The Single responsibility and Open-closed principles suggest the importance of module independence.

design rule theory explains these informal principles so that they can be visualized and possibly quantified 解释了非正式原则，并让他们可以可视化并量化的基础

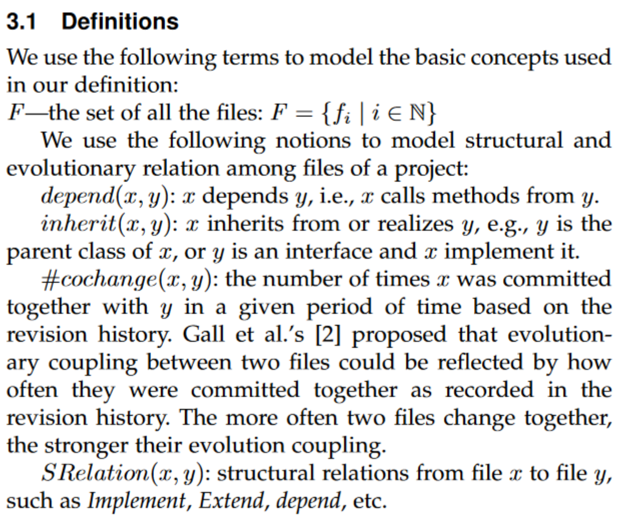
> **里氏替换原则**（Liskov Substitution principle）是对子类型的特别定义。它由芭芭拉·利斯科夫（Barbara Liskov）在1987年在一次会议上名为"数据的抽象与层次"的演说中首先提出。[1]
>
> - "派生类（子类）对象可以在程序中代替其基类（超类）对象。"

> **接口隔离原则**（英语：interface-segregation principles， 缩写：ISP）指明客户（client）不应被迫使用对其而言无用的方法或功能。
>
> - 拆分非常庞大臃肿的接口成为更小的和更具体的接口，这样客户将会只需要知道他们感兴趣的方法。这种缩小的接口也被称为**角色接口**（role interfaces）。

> - 目的是系统解开耦合，从而容易重构，更改和重新部署。接口隔离原则是在SOLID中五个面向对象设计（OOD）的原则之一，类似于在GRASP中的高内聚性。

> **依赖反转原则**（Dependency inversion principle，DIP）是指一种特定的解耦（传统的依赖关系创建在高层次上，而具体的策略设置则应用在低层次的模块上）形式，使得高层次的模块不依赖于低层次的模块的实现细节，依赖关系被颠倒（反转），从而使得低层次模块依赖于高层次模块的需求抽象。
>
> 该原则规定：
>
> 1. 高层次的模块不应依赖于低层次的模块，两者都应该依赖于抽象接口
> 2. 抽象接口不应该依赖于具体实现。而具体实现则应该依赖于抽象接口。
>
> 该原则颠倒了一部分人对于面向对象设计的认识方式。如高层次和低层次对象都应该依赖于相同的抽象接口。[1]

基本术语 and the rationale and formalization of these anti-patterns



3.1 Definitions

We use the following terms to model the basic concepts used in our definition:
F—the set of all the files: $F = \{f_i \mid i \in \mathbb{N}\}$

We use the following notions to model structural and evolutionary relation among files of a project:

$depend(x, y)$: x depends y, i.e., x calls methods from y.

$inherit(x, y)$: x inherits from or realizes y, e.g., y is the parent class of x, or y is an interface and x implement it.

$\#cochange(x, y)$: the number of times x was committed together with y in a given period of time based on the revision history. Gall et al.'s [2] proposed that evolutionary coupling between two files could be reflected by how often they were committed together as recorded in the revision history. The more often two files change together, the stronger their evolution coupling.

$SRelation(x, y)$: structural relations from file x to file y, such as *Implement*, *Extend*, *depend*, etc.

六种架构反模式：具体的看文章内部

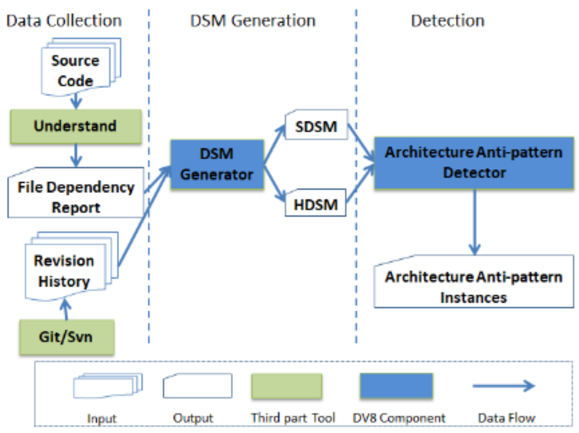
注意！！！ smell != anti-pattern

1. Unstable Interface (UIF)
2. Modularity Violation Group(MVG)
3. Unhealthy Inheritance Hierarchy(UIH)
4. Crossing(CRS)
5. Clique(CLQ)
6. Package Cycle(PKC)

经典的架构反模式检测框架，与DV8 文章高度相似，应该说抽取了一部分

[21] R. Mo, W. Snipes, Y. C. S. Ramaswamy, R. Kazman, and M. Naedele, “Experiences applying automated architecture analysis tool suites,” in Proc. 33rdIEEE/ACM International Conference on Automated Software Engineering, 2018, pp. 779–789.

Fig. 9: Framework of Architecture Anti-pattern Detection



RQ:

RQ1. Do infected files consume significantly more maintenance effort than non-infected files?受感染的文件是否比未受感染的文件消耗更多的维护工作?

RQ2. If a file is involved in greater numbers of architecture anti-patterns, then is it more error-prone/change-prone?如果一个文件涉及更多的体系结构反模式，那么它是否更容易出错/更容易更改?

RQ3. Do different architecture anti-patterns have different impacts on error-proneness and change-proneness? 不同的架构反模式对易错性和易变性有不同的影响吗?

Q：架构反模式和AS的差别?

---

2. Arcan: a Tool for Architectural Smells Detection

Fontana, F. A., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., & Di Nitto, E. (2017). Arcan: A tool for architectural smells detection. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. https://doi.org/10.1109/icsaw.2017.16

评价:

- 和我们的project内容高度相似，值得学习
- 非常短的文章，精髓浓缩在如下

Abstrat:

Code smells are sub-optimal coding circumstances such as blob classes or spaghetti code - they have received much attention and tooling in recent software engineering research.

Higher-up in the abstraction level, architectural smells are problems or sub-optimal architectural patterns or other design-level characteristics. These have received significantly less attention even though they are usually considered more critical than code smells, and harder to detect, remove, and refactor.

This paper describes an open-source tool called Arcan developed for the detection of architectural smells through an **evaluation of several architecture dependency issues.** The detection techniques inside Arcan exploit **graph database technology**, allowing for high scalability in smells detection and better management of large amounts of dependencies of multiple kinds.

In the scope of this paper, we focus on the evaluation of Arcan results carried out with **real-life software developers** to check if the architectural smells detected by Arcan are really perceived as problems and to get an overall usefulness evaluation of the tool.

翻译：codesmell 是欠佳的编程条件，如blob类或意大利面条式代码——它们在最近的软件工程研究中得到了很多关注并产生了很多针对性的工具。

而在抽象层的更高层，architecture smell 带来了（巨大的）问题，或者欠佳的architectural patterns或其他design-level 特征。尽管它们通常被认为比code smell更重要，而且更难以检测、删除和重构，但它们受到的关注却少得多。

这篇文章描述了一个名为Arcan的开源工具，其通过评估 几个架构依赖议题 来检测architecture smell。Arcan内部的检测技术利用了图形数据库技术，使smell detection具有高可扩展性，并能更好地管理多种类型的大量依赖关系。

在本文的范围内，我们将重点放在对现实软件开发人员进行的Arcan结果的评估上，以检查Arcan检测到的architecture smell是否真的被视为问题，并获得工具的总体有用性评估。

Arcan 原理

Arcan, a **static-analysi**s software useful to support software developers and designers during the development, maintenance and evolution of Java applications.

- 静态代码分析 + *JAVA*
- 只能检测3种AS Cycle Dependency, Unstable Dependency and Hub-Like Dependency
- 资料少，没有什么开源生态, 不支持docker

提问：在Designate里面如何?

提问2：package level v.s. file level?

- package就是多个类文件和他配置文件放一起的一个东西，打包嘛,然后可以用来import之类的,有jar等各种形式
- file就是一个文件吧，java一个文件就是一个类
- 就是可以理解为，JAVA这种严谨编程语言里的一些，类似结构大小的划分？可这能泛用到c++，python等语言当中吗

3AS

1. Cyclic Dependency (CD):

- detected on **classes and packages**
- refers to a subsystem (component) that is involved in a chain of relations that break the desirable acyclic nature of a subsystem’s dependency structure. 指的是一个子系统(组件)，它所涉及的关系链打破了子系统依赖结构的理想非循环性质。
- The subsystems involved in a dependency cycle can be hard to release, maintain or reuse in isolation. 依赖周期中涉及的子系统可能很难单独发布、维护或重用。
- 实践： we found useful to refine the detected cycles according to their shape [11] (shown in Figure 1).

- 他们自定义了每个形状的名字 established rules to identify each shape. Some of them are formulas which set the relationship between the number of nodes and edges; others are constraints at graph level, i.e., patterns that nodes and edges have to follow to make up a certain kind of shape.

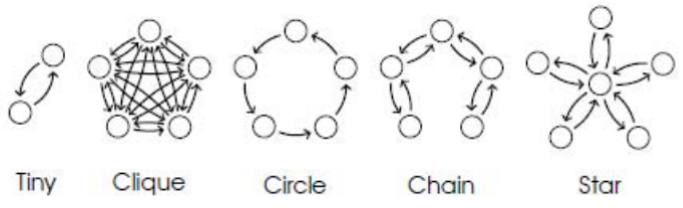
- 

Fig. 1. Cycles shapes [11]

2. Unstable Dependency (UD):

- detected on **packages**
- describes a subsystem (component) that depends on other subsystems that are less stable than itself, according to the Instability metric value [1] .This may cause a ripple effect of changes in the system.描述了一个子系统(组件)，它依赖于其他子系统，这些子系统比它本身更不稳定，根据不稳定性度量值，这可能会引起系统变化的连锁反应。
- **过滤器！** For this smell, we defined a filter to remove false positive instances. Since a package is considered affected by this smell only if it depends on another package less stable than itself, it was interesting to examine the dependencies which actually cause the smell. 只有一部分的smell是有害的
- **"Bad Dependency"**： a dependency that points to a less stable package, 公式化："Degree of Unstable Dependency" (DoUD): A package with a small number of bad dependencies may not be a smell, and this formula helps to filter misleading result. "不稳定依赖程度"(Degree of instability dependency, DoUD):带有少量不良依赖的包可能不是AS，这个公式有助于过滤误导的结果
    - $DoUD = \frac{BadDependencies}{TotalDependencies} \times 100\%$
- The Degree of Unstable Dependency under which a package is no more a smell can be defined as a **threshold** of the filter. 这些例外可以被视作过滤器的阈值。
- 目前：set at **30%, 经过纯手动测试**得出的结论，他们希望未来能用机器学习技术去确认阈值

3. Hub-Like Dependency (HL): 成为像依赖(HL):

- detected on **classes**
- this smell arises when an abstraction has (outgoing and ingoing) dependencies with a large number of other abstractions [2]. 当一个抽象与大量其他抽象具有(外向的和外向的)依赖性时，就会产生这种气味
- 提问：与star-shape CD的差别是什么？
- 吐槽：感觉这些很像数学里的拓扑学内容，或许定义上就是相同的
- For this smell, we figured out which are the conditions where a Hub-Like could be a **false positive instance**:
    - First, HL are highly used classes of the system. Assuming we don't know if those classes are used from other systems (**since Arcan analyzes one system at a time)**, we conjectured that if the class uses external classes of the system, e.g., classes of the package java.util.*, and these are the majority of the total outgoing dependencies related to a system library, then it should *not* be considered a Hub-Like class.

    - In fact, classes of this form are rather simple, because they most likely **use default functionalities** (e.g lists). Conversely, classes that are frequently used and implement the main functionalities of the system **exhibit the opposite pattern.** 实践当中特征差距很大
- 对于这种气味，我们找出了Hub-Like可能是假阳性实例的哪些条件。
- 首先，HL是系统中高度使用的类。 假设我们不知道这些类是否在其他系统中使用(因为Arcan一次分析一个系统)，我们推测如果类使用系统的外部类，例如java.util包中的类。 ，并且这些是与系统库相关的大部分输出依赖项，那么它不应该被视为HL smell。
- 事实上，这种形式的类相当简单，因为它们很可能使用默认功能(例如列表)。 相反，经常使用并实现系统主要功能的类则表现出相反的模式。

Result

将另外两个软件：DICER & Tower4Clouds 视作ground truth 后检测的结果如下

TABLE II
ARCHITECTURAL SMELLS IN THE ANALYZED COMPONENT

	DICER	Tower4Clouds
Total Architectural Smells	5	9
True Positive	3	6
False Positive	0	0
False Negative	2	3
True Negative	0	0
Precision(%)	100	100
Recall(%)	60	66
F-measure(%)	75	79,52

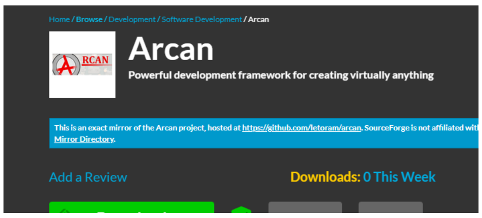
TABLE III
DETECTED ARCHITECTURAL SMELLS BY ARCAN

	DICER	Tower4Clouds
Cyclic Dependency (class)	636	439
Cyclic Dependency (package)	83	38
Unstable Dependency	305	123
Hub Like Dependency	1	3
Totals	1025	603

[[[[[待补充]]]]]

PS.

1. 千万别把arcan 和一个framework 搞混，github和docker 都没有这个软件的download Method



Home / Browse / Development / Software Development / Arcan

Arcan

Powerful development framework for creating virtually anything

This is an exact mirror of the Arcan project, hosted at https://github.com/letoram/arcan. SourceForge is not affiliated wit
Mirror Directory.

3. Architectural smells detected by tools: A catalogue proposal

Azadi, U., Arcelli Fontana, F., & Taibi, D. (2019). Architectural smells detected by tools: A catalogue proposal. *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. https://doi.org/10.1109/techdebt.2019.00027

4. What is technical debt?

What is technical debt? – Arcan

> *Technical debt is a concept in software engineering that refers to the amount of effort required to fix suboptimal code, design, and architecture.* Technical debt makes the work of your developers harder and hinders you from delivering features quickly.

Technical debt (TD) is a metaphor that uses the concept of financial debt to describe poor software quality. The two main concepts of the metaphor are *principal* and *interest*:

- **principal** is the amount of effort necessary in order to remove all the problems present in your system that prevent you from going fast;
- **interest** is the extra effort that it takes to add new features to the system because of the TD present.

When a software system is affected by technical debt, all the design-time quality attributes of the system are compromised. The system is:

- hard to maintain (e.g. fix bugs, understand, etc.)
- hard to evolve (e.g, adding new features),
- harder to secure

> When a software system is affected by technical debt, all the design-time quality attributes of the system are compromised.

What are the causes of technical debt?

There are many causes that lead to the increase of technical debt in a software system. First of all, the time-to-market pressure. When developers are required to release new features in a short amount of time they create a quick-and-dirty solution that may meet the urgent needs of the moment but it is not solid enough to provide a strategical advantage in the long run.

It is of paramount importance that the quick-and-dirt solution is fixed at the first available moment.

> time-to-market pressure is one of the most common causes of technical debt.

Another common source of TD is the adoption of overly complex design. If the code is too difficult to comprehend due to intricate design constructs and patterns, adding new features may take an excessive amount of time to develop. This, together with a lack of system documentation, can be more harmful than actual bad design.

Last but not the least, delayed refactorings can be one of the causes. Refactoring is a term indicating the actions needed to fix a design problem while preserving the expected software behavior. A good practice to avoid the growth of technical debt is to schedule refactoring from the very early phases of software development.

> Refactoring is a term indicating the actions needed to fix a design problem while preserving the expected software behavior.

How can you prevent the accumulation of technical debt?

Technical debt *naturally* accumulates in software systems, however, planned refactorings to fix quick-and-dirty solutions and the careful monitoring of its evolution can prevent most of the deterioration.

Arcan integrates with Continuous Integration (CI) pipelines and allows to continuously detect and monitor technical debt. Ask us for a trial to learn how to prevent technical debt.

---

1. R. C. Martin, “Object oriented design quality metrics: An analysis of dependencies,” ROAD, vol. 2, no. 3, Sept–Oct 1995. ↩
2. G. Suryanarayana, G. Samarthyam, and T. Sharma, Refactoring for Software Design Smells, 1st ed. Morgan Kaufmann, 2015. ↩