SCUOLA DI DOTTORATO
UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

Department of

Informatics Systems and Communication

PhD program Computer Science Cycle XXX

IDENTIFYING AND EVALUATING SOFTWARE ARCHITECTURE EROSION

Surname Roveda Name Riccardo

Registration number 723299

Tutor: Prof. Alberto Ottavio Leporati

Supervisor: Prof. Francesca Arcelli Fontana

Coordinator: Prof. Stefania Bandini

ACADEMIC YEAR 2016/2017

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API	Application Programming Interface
AS	Architectural Smell (See Chapter 2)
TD	Technical Debt (See Section 2.5)
TDI	Technical Debt Index (See Section 2.5)

1 INTRODUCTION

It is an established fact that good software architecture and design lead to better evolvability, maintainability, availability, security, software cost reduction and more [24]. Conversely, when that architecture and design process are compromised by poor or hasted design choices, the architecture is often subject to different architectural problems or anomalies, that can lead to software faults, failures or quality downfalls such as a progressive architecture erosion [58, 144].

Van Gurp et al. said that systems erode over time when they evolve [57] bringing architectural erosion and degradation [58] to the applications, and the challenge to keep the intended architecture aligned with code is still hard when software engineers must deal with obsolescence and maintenance. In order to combat obsolescence and rigidity of designs, the metaphor of Technical Debt (TD) introduced by Cunningham [42] to explain the causes and need of refactoring covers an important role for software maintenance. As defined by Suryanarayana et al. [149], Design debt is concerned with the design evolution and trade-off decisions made during the architecting and implementation phases, and has to be repaid during maintenance. Today, the ways for identifying the most visible TD symptoms are several [75, 89, 118], e.g., smells, metrics and violations of different kings and at different levels. Software designers must identify and estimate the size of the debt to be repaid to yield more sustainable architectures [77]. As stated by Letouzey and Ilkiewicz [86], there is no exact method to estimate TD because its calculation should be based on all the violations to be fixed. Nevertheless, the cost of fixing design violations sometimes exceeds the allocated budget. Thus, the estimated cost of the debt is often based on the severity of the design/code violations and the cost of their repair.

Most methods attempt to estimate the debt and software quality based on source code analysis, often related to specific quality characteristics [66], but there is a need for more approaches focused on identifying and estimating the debt at architectural level. Different tools provide support in this direction, even if they often offer too much information. For example, a huge number of different quality metrics is often reported. Their interpretation is difficult, and deciding how to improve the system by simply relying on the metrics values is very hard. Moreover, many of the proposed views are often completely incomprehensible. It would be also useful if the tools could provide some kind of prioritization of the problems to be removed and suggestions to improve the architectural degradation. Among the different problems, we can identify, e.g., code smells, architectural smells and possible relations among them that could be investigated to find the patterns of smells leading to architectural problems [7, 99].

Code smells have been defined by Fowler et al. [48] as symptoms of possible problems that can be removed through refactoring steps. Other definitions of smells have been proposed in the literature, as for example those of Lanza and Marinescu [80], of Kerievsky [73], and Wake [160].

Code smells are problems relying code level, while architectural smells (AS) are their natural counterpart at architectural level. As defined by Garcia et al. [52], "An Architectural smell is a commonly used architectural decision that negatively impacts system quality. Architectural smells may be caused by applying a design solution in an inappropriate context, mixing design fragments that have undesirable emergent behaviours, or applying design abstractions at the wrong level of granularity."

A first definition of Architectural Smell

However this definition rises two questions: *are both internal and external quality considered? Which decisions cause architectural smells?* Trying to answer this questions, I propose an enhanced AS definition: "An architectural smell is a commonly used architectural decision, intentional or not intentional, that negatively impact *internal* system quality". Hence, architectural smells can have a large effect on software maintainability [93].

My definition of Architectural Smell

For what concerns code smells and metrics, many tools have been developed, both open source and commercial ones. Architectural smells received less attention, even if some tools have been developed [17], see Section 2.2.

1.1 MAIN CONTRIBUTIONS OF THE THESIS

The main contributions of the thesis are: *1)* the development of Arcan, a tool for architectural smells detection on Java projects, *2)* the validation of Arcan results, *3)* different empirical analysis on architectural smells related to architectural smells prediction, and the possible correlations existing between architectural smells and code smells, *4)* the definition of a new architectural smells index.

Before introducing Arcan I report three experiences, described in Chapter 3, which aim to evaluate architectural erosion through different available tools.

The first experience [11] was performed using three known tools to capture information that can be useful to identify and evaluate the architectural debt of an application; I outlined the main differences among these tools and the results they produce by analyzing a project. The second experience [16] has been done using two tools to analyze 2 projects in order to detect problems that could cause architecture degradation, and in repairing some of these problems directly through the support of the tools, e.g., by removing cyclic dependencies, or by manually changing the code to remove some architectural violations. The third experience consisted in repairing and removing architectural smells and antipatterns on 4 open source projects [18], by monitoring the impact on different Technical Debt (TD) indexes using four tools.

In Chapter 4, I described the design of Arcan and explained its architecture. Since Arcan analyzes compiled Java projects, Arcan reconstructs them from class files. Arcan represents the analyzed projects through their dependency graph which can be stored using a graph database. Moreover, the graph could be useful to identify possible refactoring opportunities of an architectural smell. I defined the algorithms to detect the architectural smells and I focused my attention on the detection of six dependency-based architectural smells, i.e., architectural smells detected using the dependency graph. I exploited techniques to detect one architectural smell using the project' development history data obtained through the Versioning Control Systems. This smell is called Implicit Cross Package Dependency and reveals hidden dependencies (such as co-changes) among files which are not belonging to the same package. Finally, the tool has been validated through three different case studies [17, 19, 135] described in Chapter 5.

In Chapter 6, I studied the development history of 4 projects. By applying machine learning techniques, I identified that it is possible to predict architectural smells by using the architectural smells which are present in the past. I discovered that changes made in past can lead to introduction of new architectural smells, while other factors like the number of developers do not lead to the introduction of a new architectural smell. The study of the evolution of architectural smells overtime has shown an increasing trend and a visual correlation was discovered among different types of architectural smell during the development history. Moreover, I conducted a large-scale empirical study to investigate the relations between code smells and architectural smells, and to understand if code smells affect the presence of architectural smells and vice versa. I found empirical evidence on the independence between code smells and architectural smells [135] and therefore, I can formulate that the presence of code smells does not imply the presence of architectural smells.

In Chapter 7, I proposed and validated a new Architectural Debt Index (ADI) focused on the evaluation of the severity of the architectural smells in a project, as outlined in [10]. The index computation takes into account *a)* the number of architectural smells in a projects, *b)* the severity of the architectural smells, which is defined according to their type and *c)* the history of the smells, which indicates the impact of the architectural smell presence. The ADI can be used to compare different projects and their quality during their evolution. I also proposed a profile-oriented ADI computation, based on the ISO/IEC 25010:2011 standard [65], where the architectural smells can be grouped according to the affected quality attribute. I outlined that the ADI can be used to identify the most critical classes or packages in the projects; in this way, the developer/maintainer can easily identify and focus his attention on the most critical classes or packages.

In Chapter 8, I described a preliminary case study on architectural smells refactoring that allowed to identify a series of refactoring opportunities for

the architectural smells detected in a project and allowed to study the impact of the performed refactoring on selected quality metrics and indexes.

1.2 RESEARCH QUESTIONS

In the thesis I aim to answer the following Research Questions:

- **Arcan Results Validation**
  According to architectural smell detection validation described in Chapters 4 and 5, from a first validation of the tool in two industrial projects, I received a positive evaluations and useful feedbacks from the developers of Arcan. Hence, I tried to answer these research questions:

  RQ1 *Is Arcan exhaustive in architectural smells detection?*

  RQ2 *Is Arcan correct in its architectural smells detection?*

- **Empirical Analysis with Arcan**
  According to the empirical studies described in Chapter 6, I answered some research questions related to the prediction of architectural smells and another related to the possible correlations between code smells and architectural smells:

  RQ3 *Can we use the presence of architectural smells in the project's history to predict the presence of architectural smells in the future?*
  According to this topic I answered other three *RQs* described in detail in Chapter 6.1.

  RQ4 *Is the presence of an architectural smell independent from the presence of code smells?*
  According to this topic I answered other three *RQs* described in detail in Chapter 6.2.

- **Technical Debt Index**
  According to the index described in Chapter 7, I tried to answer some research questions in order to define a new Architectural Debt Index:

  RQ5 *How should a new index be formulated to better represent the architectural debt?*
  According to this topic I answered other two *RQs* described in detail in Chapter 7.

- **Architectural Smell Refactoring**
  According to the refactoring of architectural smells, I tried to answer a research question related to the detection of refactoring recommendation opportunities of architectural smell:

  RQ6 *How often can refactoring opportunities and/or recommendation be identified for architectural smells?*
  According to this topic I answered other three *RQs* described in detail in Chapter 8.

1.3 PUBLICATIONS

The lists of published, submitted or unsubmitted papers with the linked RQs are reported in the following.

1.3.1 *Published papers*

I. Tollin, F. Arcelli Fontana, M. Zanoni, and **R. Roveda**, "Change Prediction through Coding Rules Violations", *in Proceedings of the* 21*st International Conference on Evaluation and Assessment in Software Engineering*, EASE 2017, Karlskrona, Sweden, June 15-16, 2017, pp. 61–64.

RELATED RQ: ***RQ3***

F. Arcelli Fontana, I. Pigazzini, **R. Roveda**, D. A. Tamburri, M. Zanoni, and E. D. Nitto, "Arcan: A Tool for Architectural Smells Detection", *in* 2017 *IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017*, Gothenburg, Sweden, April 5-7, 2017, pp. 282–285.

RELATED RQ: ***RQ1***, ***RQ2***

F. Arcelli Fontana, **R. Roveda**, M. Zanoni, C. Raibulet, and R. Capilla, "An Experience Report on Detecting and Repairing Software Architecture Erosion", *in* 13*th Working IEEE/IFIP Conference on Software Architecture*, WICSA 2016, Venice, Italy, April 5-8, 2016, pp. 21–30.

RELATED RQ: ***RQ6***

F. Arcelli 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 2016, Raleigh, NC, USA, October 2-7, 2016, pp. 433–437.

RELATED RQ: ***RQ1***, ***RQ2***

F. Arcelli Fontana, **R. Roveda**, S. Vittori, A. Metelli, S. Saldarini, and F. Mazzei, "On evaluating the impact of the refactoring of architectural problems on software quality", *in Proceedings of the Scientific Workshop Proceedings of XP2016*, Edinburgh, Scotland, UK, May 24, 2016, p. 21.

RELATED RQ: ***RQ6***

F. Arcelli Fontana, **R. Roveda**, and M. Zanoni, "Technical Debt Indexes Provided by Tools: A Preliminary Discussion", *in* 8*th IEEE International Workshop on Managing Technical Debt*, MTD 2016, Raleigh, NC, USA, October 4, 2016, pp. 28–31.

RELATED RQ: ***RQ5***

F. Arcelli Fontana, **R. Roveda**, and M. Zanoni, "Tool support for evaluating architectural debt of an existing system: an experience report", *in Proceedings of the* 31*st Annual ACM Symposium on Applied Computing*, Pisa, Italy, April 4-8, 2016, pp. 1347–1349.

F. Arcelli Fontana, V. Ferme, M. Zanoni, and **R. Roveda**, "Towards a prioritization of code debt: A code smell Intensity Index", *in 7th IEEE International Workshop on Managing Technical Debt*, MTD@ICSME 2015, Bremen, Germany, October 2, 2015, pp. 16–24.

RELATED RQ: ***RQ5***

1.3.2 *Submitted papers*

The list of submitted papers:

R. Roveda, F. Arcelli Fontana, I. Pigazzini, M. Zanoni and P. Avgeriou, "A Study on Architectural Smells Prediction and Evolution", *Submitted to Research Track of* 34*th IEEE International Conference on Software Maintenance and Evolution*, (ICSME 2018), Madrid, Spain, 2018, IEEE.

RELATED RQ: ***RQ3***

F. Arcelli Fontana, **R. Roveda**, V. Leonarduzzi and D. Taibi, "Are Architectural Smells Independent from Code Smells? An Empirical Study", *Submitted Journal of Information and Software Technology (IST)*, 2018, Elsevier.

RELATED RQ: ***RQ4***

R. Roveda, F. Arcelli Fontana, I. Pigazzini and M. Zanoni, "Towards an Architectural Debt Index", *Submitted to TechDebt* 2018 *International Conference on Technical Debt, co-located with ICSE* 2018, (TechDebt 2018), 2018, Gothenburg, Sweden, ACM SIGSOFT/IEEE TCSE.

RELATED RQ: ***RQ5***

1.3.3 *To be submitted papers*

List of papers that have to be submitted:

R. Roveda, F. Arcelli Fontana and D. A. Tamburri, "Automated detection and Evaluation of Architectural Smells: a Mixed-Methods Study", *To be submitted*.

RELATED RQ: ***RQ1***, ***RQ2***

R. Roveda, F. Arcelli Fontana, M. Zanoni, "Refactoring of Architectural Smells Detected with Arcan: Lessons Learned", *To be submitted*.

RELATED RQ: ***RQ6***

1.3.4 *Published papers not strictly related to the thesis*

The list of papers not directly related to the main topics of the thesis:

F. Arcelli Fontana, **R. Roveda**, and M. Zanoni, "Discover Knowledge on FLOSS Projects Through RepoFinder", *in KDIR* 2014 - *Proceedings of the*

International Conference on Knowledge Discovery and Information Retrieval, Rome, Italy, 21 - 24 October, 2014, pp. 485–491.

F. Arcelli Fontana, **R. Roveda**, and M. Zanoni, "A System for the Discovery and Selection of FLOSS Projects", *ERCIM News*, vol. 2014, no. 97, 2014.

F. Arcelli Fontana, P. Braione, **R. Roveda**, and M. Zanoni, "A Context-Aware Style of Software Design", *in* 2*nd IEEE/ACM International Workshop on Context for Software Development*, CSD 2015, Florence, Italy, May 19, 2015, pp. 15–19.

R. Roveda, F. Arcelli Fontana, C. Raibulet, M. Zanoni, and F. Rampazzo, "Does the Migration to GitHub Relate to Internal Software Quality?", *in ENASE* 2017 - *Proceedings of the* 12*th International Conference on Evaluation of Novel Approaches to Software Engineering*, Porto, Portugal, April 28-29, 2017, pp. 293–300.

2

RELATED WORK

In this section I will introduce the related work on the different areas/topics addressed in my thesis:

- Architectural Smells Definitions;
- Architectural Smells Detections;
- Code Smell and Architectural Smells Correlations
- Technical Debt Indexes;
- Architectural Smells Refactoring.

2.1 ARCHITECTURAL SMELLS DEFINITIONS

Different AS have been defined in the literature by different authors as previously outlined. In this section, it is provided a short definition of all the architectural smells that I found in the literature. In most of the works, the authors refer to this type of smell as architectural smell, in one case they are called design smells [149], Model View Controller smells [4] or Hotspot pattern [114].

Architectural Smells of Lippert at al.

Lippert et al. [93] classified 30 architectural smells into the following five categories: smells in Dependency Graphs, in Inheritance Hierarchies, in Packages, in Subsystems, and in Layers. They essentially considered dependency and inheritance issues and aspects related to dimension with respect to packages, subsystems and layers. In particular, they defined the following AS:

- *Smells in Dependency Graph* are related to classes coupled through *use* dependency. The use relations among the classes of the system are related to the static dependency graph. In the following the architectural smells of this category:
  - *Obsolete classes*: classes that are no longer in use will burden the system with obviously obsolete functionality. Not only single classes can be no longer in use, but also entire dependency graphs whether all the classes on the entire dependency graphs are not used.
  - *Tree-like Dependency Graph*: tree-like dependency graphs indicate a functional decomposition of the system. Each class of the tree is used by exactly one other class. Reuse does not happen.

  - *Static Cycle in Dependency Graph*: two classes using each other constitute the simplest imaginable cycle in a dependency graph. More complex cycles can also include various classes.
  - *Visibility of Dependency Graph*: it arise when the principles of encapsulation and of information hiding is not applied correctly, i.e., the internal implementation is hidden behind an interface. A system with a public dependency graph will create more problems if one tries to change it, whereas changes to a private dependency graph will only have local effects.

- *Smells in Inheritance Hierarchies* are related to Object-Oriented (OO) inheritance coupling between classes. The coupling via inheritance provides the advantage of polymorphism (the ability to adopt multiple shapes). During run-time, objects of different types can stand behind one identifier. This is made possible through polymorphic assignments. If an object is bound to a variable, this object must not necessarily possess this particular variable type. It is sufficient if the object's type is a subtype of the variable type. Inheritance also results in a closer coupling than use. This is why inheritance hierarchy problems are quite severe: due to the close coupling of the classes in the hierarchy, problems will be passed on from superclasses to their subclasses. In the following the architectural smells of this category:
  - *Type queries*: the inheritance relation expresses itself not only in the classes of the inheritance hierarchy, but in the clients too. It violate the Once and Only Once principle.
  - *List-like Inheritance Hierarchy*: in a list-like inheritance hierarchy each class possesses a maximum number of one subclass. Such inheritance hierarchies either point to speculative generalizations or to too big classes. Speculative generalization means that superclasses were implemented for a definitely required class in hopes that the created abstraction might come in handy later on. This situation occurs quite often when the class hierarchy only consists of two classes.
  - *Subclasses Do Not Redefine Methods*: if subclasses don't redefine methods of their superclass, this can indicate that no abstraction is expressed through inheritance and pure implementation inheritance. Often a uses relation between classes will turn out to be more effective.
  - *Inheritance Hierarchies Without Polymorphic Assignments*: similar to the *Subclasses Do Not Redefine Methods* smell, inheritance hierarchies without their respective polymorphic assignments point to the presence of unnecessary generalizations. The most significant advantage of inheritance as opposed to use is its flexibility, which is achieved through polymorphism. If no polymorphic assignments exist, this flexibility will not be used, and inheritance can be replaced by uses relations.

  - *Parallel Inheritance Hierarchies*: when two or more classes inheritance hierarchies grow together because they are dependent. For example, an existing domain-specific inheritance hierarchy between the business objects Partner, Customer and Supplier. Partners, customers and suppliers should be displayed on the UI level in list form. Thus, one view class exists for each of the three business object classes. These view classes inherit from each other according to the business object classes' hierarchy.
  - *Too Deep Inheritance Hierarchy*: deep inheritance hierarchies can result in extremely flexible systems. Unfortunately, at the same time the system's understandability and the adaptability of its inheritance hierarchies suffers. If inheritance takes place across 10 levels, it is almost impossible to determine which implementation of a method is called by reading the code.

- *Smells in Packages* (e.g. smell related to Java packages):
  - *Unused Packages*: packages that are not in use burden the system with clearly obsolete functionality
  - *Cycles between Packages*: cycles between packages can be created through use, inheritance or through a combination of use and inheritance
  - *Too Small Packages*: packages with one or two classes are often not worth the effort of introducing them: the complexity created by the package is not offset by its additional structuring. Such too small packages can easily be removed through relocation of their classes to other packages. However, one must make sure that in this process no new cycles between packages are created.
  - *Too Large Packages*: packages with a high number of classes can be handled much easier if they are broken down into several subpackages. This will especially lead to their better understandability. Sometimes too large packages indicate missing subsystems. The creation of a subsystem from a too large package can solve this problem – for instance, if one splits the initially too large package into an interface package and one or more implementation packages.
  - *Package Inheritance Unbalanced*: similarly to inheritance hierarchies, shallow package hierarchies are easier understandable than deep ones.
  - *Package Not Clearly Named*: especially packages containing classes that are not domain-oriented are often named ambiguously, and assigning of identical names occurs. Ambiguously named packages frequently indicate that the developers had no real understanding of what's inside the packages, so it will come as no surprise if such packages contain classes with workarounds or were simply created by mistake.

- *Smells in Subsystems* (i.e., a collection of packages):
  - *No Subsystems*: from a certain size on, a system's structure – if it is exclusively defined on the package level – will become increasingly incomprehensible. If the system consists of more than 100 packages, for example, it is extremely difficult to recognize and define the structure between the packages and to maintain it consistently.
  - *Subsystem Too Large*: the subsystem are incomprehensible and containing too many concerns. In many cases, the occurrence of very large subsystems is accompanied by a loss of clarity: the subsystem is no longer responsible for a single task, but it also takes on concerns in other areas.
  - *Subsystems Too Small*: too small subsystems shift complexity from subsystems into the dependencies among the subsystems themselves. In the most extreme case, each class represents its own subsystem. Obviously this will not lead to a reduction of complexity, instead developers are confronted with an impracticable tangle of dependencies between subsystems.
  - *Too Many Subsystems*: if a system consists of many more than 30 subsystems without further grouping, the understandability of the system will be seriously impaired. This many subsystems and their interrelations can no longer be handled.
  - *Subsystems-API Bypassed*: the abstract public interface are defined to access and use a subsystem. Experience shows that such conventions are bypassed under pressure, e.g. lack of project time – either by mistake or on purpose. Bypassing the subsystem-API and directly accessing the internal implementation of the component is a practice that is not only common, but also potentially fatal.
  - *Subsystems-API Too Large*: when the API of subsystem becomes too large in relation to the implementation, the main purpose of the subsystems is not served. A major part of the system will be visible to all other subsystems. Therefore, no significant complexity reduction has been achieved.
  - *Cycles Between Subsystems*: cycles between subsystems can be created via use, inheritance or through a combination of use and inheritance
  - *Overgeneralization*: In order to assure that subsystems provide the greatest extent of reusability, they must be flexibly applicable. This generalization can be overdone though, which will result in the subsystem's over-generalization. It will become more flexible than it actually needs to be. Not only does this lead to additional subsystem development work; it also makes using the subsystem more difficult. Over-generalization occurs when the clients – in re-

lation to the size of the used subsystems – require a large amount of code.

- *Smells in Layers*, ordered group of subsystems:
  - *No Layers*: layers are not defined and there is no ordering between the subsystems.
  - *Cycles (Upward References) between Layers*: if a layer uses a higher located layer, the basic principle of layering has been ignored.
  - *Strict Layers Violated*: since the common programming languages do not provide concepts for the definition of layers, layers must be built based on conventions. In this scenario, one cannot reliably prevent that strict layers are violated. It can always happen that a layer skips the one directly beneath it and accesses a layer further below instead, be it accidentally or on purpose
  - *Inheritance between Protocol-oriented Layers*: inheritance between protocol-oriented layers is not allowed. Otherwise a stricter than desirable coupling would occur. In particular it would become impossible to re-implement the layer that inherited in a non-object-oriented programming language later on. Moreover, inheritance generally restricts the alterability of the lower layer, because changes to the superclasses can only to a certain extent be hidden from subclasses
  - *Too Many Layers*: the existence of many layers can create unnecessary indirections: one indication of unnecessary indirections are dumb delegations: one method simply invokes another method without implementing any functionality of its own.
  - *References between Vertical Separated Layers*: layers can be arranged also vertically. This is often done to structure separate products or business sections. For example, a product line is a set of software systems that share a common basis. Besides using the same basis, no further references between these systems are allowed. References between vertically separated layers create dependencies between layers. Thus the purpose of product lines *a)* Delivery, a vertical layers shall be deliverable and applicable independently from each other; *b)* Parallel development, for each single vertical layer one team shall be responsible, which does not have to confer with other layer teams regarding changes.

They essentially considered dependency and inheritance issues and aspects related to small/large size respect to packages, subsystems and layers.

Architectural Smells of Suryanarayana et al.

Suryanarayana et al. [149] classify a number of recurring structural design smells based on how they violate key Object-Oriented (OO) design principles of the PHAME "design principles" of Booch's object model [29] (Principle of Hierarchy, Abstraction, Modularity and Encapsulation). Based on

that model, they defined and organized design smells in 4 categories: abstraction smells, encapsulation smells, modularization smells and hierarchy smells. In the following, each category and the contained design smells are described.

Abstraction Smells These smells violate techniques for applying the principle of abstraction, e.g., ensure coherence and completeness, assign single and meaningful responsibility, and avoid duplication. The principle of abstraction advocates the simplification of entities through reduction and generalization: reduction is by elimination of unnecessary details and generalization is by identification and specification of common and important characteristic.

- *Duplicate Abstraction* arises when two or more abstractions have identical names, identical implementations, or both.
- *Imperative Abstraction* arises when an operation is turned into a class;
- *Incomplete Abstraction* arises when an abstraction does not support complementary or interrelated methods completely;
- *Missing Abstraction* arises when clumps of data or encoded strings are used instead of creating a class or an interface;
- *Multifaceted Abstraction* arises when an abstraction has more than one responsibility assigned to it;
- *Unnecessary Abstraction* occurs when an abstraction which is actually not needed (and thus could have been avoided) is introduced in a software design;
- *Unutilized Abstraction* arises when an abstraction is left unused (either not directly used or not reachable);

Encapsulation Smells These smells violate the encapsulation principle, e.g., hiding implementation details and hiding variations. The principle of encapsulation advocates separation of concerns and information hiding through techniques such as hiding implementation details of abstractions and hiding variations.

- *Deficient Encapsulation* occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required;
- *Leaky Encapsulation* arises when an abstraction "exposes" or "leaks" implementation details through its public interface;
- *Missing Encapsulation* occurs when the encapsulation of implementation variations in a type or hierarchy is missing;

- *Unexploited Encapsulation* arises when client code uses explicit type checks (using chained if-else or switch statements) instead of exploiting the variation in types encapsulated within a hierarchy.

Modularization Smells These smells violate techniques for applying the principle of modularization, e.g., localize related data and methods, decompose abstractions to manageable size, create acyclic dependencies and limit dependencies. The principle of modularization advocates the creation of cohesive and loosely coupled abstractions through techniques such as localization and decomposition.

- *Broken Modularization* 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* arises when an abstraction (such as a class or interface) exists that has not been completely decomposed and a further decomposition could reduce its size, implementation complexity, or both;
- *Cyclically-Dependent Modularization* arises when two or more abstractions depend on each other directly or indirectly, creating a tight coupling between the abstractions;
- *Hub-like Modularization* arises when an abstraction has dependencies (both ingoing and outgoing) with a large number of other abstractions.

Hierarchy Smells These smells violate techniques for defining proper abstraction hierarchies, e.g., apply meaningful classification, ensure substitutability, avoid redundant paths, and ensure proper ordering. The principle of hierarchy advocates the creation of a hierarchical organization of abstractions using techniques such as classification, generalization, substitutability and ordering.

- *Missing Hierarchy* arises when a code segment uses conditional logic to explicit manage variation in behaviour, where a hierarchy could have been created and used to encapsulate those variations;
- *Unnecessary Hierarchy* arises when the whole inheritance hierarchy is unnecessary, indicating that inheritance has been applied needlessly for the particular design context;
- *Unfactored Hierarchy* arises when there is unnecessary duplication among types in hierarchy;
- *Wide Hierarchy* arises when an inheritance hierarchy is "too" wide indicating that some intermediate types may be missing;
- *Speculative Hierarchy* arises when one or more types in a hierarchy are provided speculatively, for future use;

- *Deep Hierarchy* arises when an inheritance hierarchy is "excessively" deep;
- *Rebellious Hierarchy* arises when a subtype rejects the methods provided by its supertype;
- *Broken Hierarchy* arises when a supertype and its subtype conceptually do not share an "IS-A" relationship resulting in broken substitutability;
- *Multipath Hierarchy* arises when a subtype inherits both directly and indirectly from a supertype, leading to unnecessary inheritance paths in the hierarchy;
- *Cyclic Hierarchy* arises when a supertype in a hierarchy depends on any of its subtypes.

Architectural Smells of Macia

Macia [98] analysed different architectural smells, as those defined in the following:

- *Ambiguous Interface*: The interface offers only a single general entry-point and, hence, it can handle more requests than it should actually process.
- *Component Concern Overload* (Component Responsibility Overload): A component is responsible for realizing two or more unrelated systems concerns.
- *Connector Envy*: A component realizes functionality that should be assigned to a connector.
- *Cyclic Dependency*: A relation between two or more architectural elements that depend on each other either directly or indirectly.
- *Extraneous Connector*: occurs when two connectors of different types are used to link a pair of components. The problem is that the beneficial effects of each individual connector may cancel each other out. This smell is also known as Extraneous Adjacent Connector [52].
- *Feature concentration*: opposes the properties in the Scattered Functionality smell by implementing different functionalities in a single design construct. Such decision might also be accompanied by the definition of only one generic entry point for a component, indicating the occurrence of the ambiguous interfaces smell.
- *Overused Interface*: Interface that requires a lot of data or is required by several interfaces.
- *Redundant Interface*: Interface that requires the same information of other interfaces.
- *Scattered (Parasitic) Functionality*: A high-level concern is spread over multiple modules that implement different concerns.

Architectural Smells of Garcia et al.

Garcia et al. [52] defined the Connector Envy, Scattered Functionality, Ambiguous Interface and Extraneous Connector also defined by Macia above. They provide a description of each AS, with a section on the Quality Impact and Trade-offs and a generic schematic view of each smell captured in one or more UML diagrams. They assert that architects can manually use such diagrams to inspect their own designs to look for architectural smells.

Architectural Smells of Kazman at al.

Kazman et al. [72] and Mo et al. [114] have defined five AS, four defined at file level and one at package level, that they call Hotspot Patterns. These AS have been defined in the context of the authors' research on Design Rule Spaces (DRSpaces) [162].

- *Unstable Interface*: when a leading file is changed frequently with other files in the revision history.
- *Implicit Cross-module Dependency*: based on the concept of true modules, this pattern aims to reveal hidden dependencies connecting modules that appear to be mutually independent. Files belong to different independent modules in the Design Rule Hierarchy clustering, but are changed together frequently. This pattern represents a kind of modularity violation.
- *Unhealthy Inheritance Hierarchy*: it detects hierarchical structures that violate either design rule theory or the Liskov Substitution principle.
- *Cross-Module Cycle*: when there is a dependency cycle and not all the files belong to the same module; the smell addresses a proper definition of a hierarchy structure among modules.
- *Cross-Package Cycle*: related to forming a proper hierarchy structure among packages. A cycle among packages is typically considered to be harmful.

Implicit Cross-module Dependency and Unstable Interfaces cannot be detected by examining a single version of a code base, but its history.

Architectural Smells of Marinescu

Marinescu et al. [106, 108] defined three AS:

- *Cyclic Dependency* refers to a subsystem (component) that is involved in a chain of relations that break the desirable acyclic nature of a subsystems dependency structure. The subsystems involved in a dependency cycle can be hardly released, maintained or reused in isolation.

- *Stable Abstraction Breaker* is a subsystem (component) for which its stability level is not proportional with its abstractness. This usually means that the incoming dependencies that make a subsystem stable are not directed towards interfaces and abstract classes, but rather towards concrete implementation classes. As a result, such subsystems are in a "zone of pain" in which it becomes impossible to extend subsystems without changing them.

- *Unstable Dependency* describes a subsystem (component) that depends on other subsystems that are less stable than itself. In this context, stability refers to the balance between the incoming and outgoing dependencies of a subsystem: many incoming dependencies make the subsystem more stable, while a large number of outgoing dependencies increase the chances that the subsystem may have to change. Subsystems affected by this flaw may cause a ripple effect of changes in the system.

Architectural Smells of Le et al.

Le et al. [83, 84] proposed the detection techniques and a classification framework of architectural smells composed by 11 architectural smells and 4 categories: concern-based smells, dependency-based smells, interface-based smells and coupling-based smells. 7 architectural smells are new, since the other AS were defined previously by Macia and Garcia.

- *Interface-Based Smell*, which emerge from carelessly defined component interfaces:
  - *Unused Interface* is an interface of a component that is linked with no other components. That interface adds unnecessary complexity to a software system which, in turn, hinders software maintenance.
  - *Unused Brick* is a component, whose interfaces are all unused interfaces. Similar to Unused Interface smell, Unused Brick adds unnecessary complexity to a system which, in turn, hinders maintenance.
  - *Sloppy Delegation* occurs when a component delegates to another component a small amount of functionality that it could have performed itself. In particular, a component could perform that functionality itself if all the data it needs to perform that functionality is part of that component's state. This inappropriate separation of concerns complicates the functionality of the system which, in turn, hinders maintenance of that system.
  - *Brick Functionality Overload* occurs when a component performs an excessive amount of functionality. Excessive functionality is another form of inappropriate modularity in a software system, which violates the principles of separation of concerns and isolation of change.

  - *Lego Syndrome* occurs when a component handles an extremely small amount of functionality. This smell represents components that are excessively small and do not represent an appropriate separation of concerns. Then, such components should be moved into another component.

- *Change-Based Smells* refers to a sets of components changing together:
  - *Duplicate Functionality* affects a component if the component shares the same functionality with other components. Changing one instance of the functionality without changing the others may create errors. Duplicated functionality increases complexity by causing any changes to one instance of the functionality to possibly require changes in the other instances.
  - *Logical Coupling* occurs when a part of a component has logical-coupling with another part in another component. Two parts have logical-coupling if they are frequently changed together in many code commits of developers. Logical-coupling could be identified by mining commit logs. This smell also increases system complexity like Duplicate Functionality.

Architectural Smells of Aniche et al.

Aniche et al. [4] provided a catalogue of six smells that are specific to web systems that rely on the MVC pattern [78] (Model View Controller pattern), where *Controllers* are classes for responsible to control the flow between the view and the model layers. Commonly, these classes represent an endpoint for other classes, do not contain state, and manage the control flow. Besides being possibly affected by "traditional smells" (e.g., God Classes), good programming practices suggest that Controllers should not contain complex business logic and should focus on a limited number of services offered to the other classes. Similarly, Data Access Object (DAO) classes [49] in MVC applications are responsible for dealing with the communication towards the databases. These classes, besides not containing complex and long methods (traditional smells) should also limit the complexity of SQL queries residing in them. The defined smells are:

- *Brain Repository*: the smell is defined as "Complex logic in the repository". Repositories are meant to deal with the persistence mechanism, such as databases. However, when Repositories contain complicated (business) logic or even complex queries, it is considered a smelly class.

- *Fat Repository*: the smell is defined as "a Repository managing too many entities". Commonly, there is a one-to-one relation between an Entity and a Repository, e.g., the entity Item is persisted by Item-repository. If a Repository deals with many entities at once, this may imply low cohesion and may make maintenance harder.

- *Promiscuous Controller*: the smell is defined as "Controllers offering too many actions". Controllers should be lean and provide cohesive operations and endpoints to clients.
- *Brain Controller*: the smell is defined as "Controllers with too much flow control". In Web MVC applications, Entities and Services should contain all business rules of the system and manage complex control flow. Even if a Controller contains just a few routes (i.e., is not a Promiscuous Controller), it can be overly smart.
- *Laborious Repository Method*: the smell is defined as "a Repository method having multiple database actions". As a good practice, a method should have only one responsibility and do one thing. Analogously, if a single method contains more than one query (or does more than one action with the database), it may be considered too complex or non-cohesive.
- *Meddling Service*: the smell is defined as "Services that directly query the database". Services are meant to contain business rules and/or to control complicated business logic among different domain classes. However, they should not contain SQL queries.

In my thesis I considered 6 architectural smells: some are already defined in the literature, but I detect them with different techniques.

2.2 ARCHITECTURAL SMELLS DETECTION

As reported in Section 2.1, AS have been defined by different authors, in some cases the definitions of the AS are very similar but not identical, leading to different names for the AS and different detection strategies. Architectural smells have received significantly less attention even though they are usually considered more critical than code smells, and harder to detect, remove, and refactor. In this section, the attention is focused on the detection of the architectural smells.

Tools have been developed, but some of them are prototypes or not yet publicly available. There are also commercial tools, e.g., Sotograph[1], Sonargraph[2], Structure101[3], CAST[4] which are able to detect architectural violations of different types, where some of them correspond to specific architectural smells, e.g., in the case of Cyclic Dependencies.

The commercial tool inFusion[5], no more available, and its evolution in Ai Reviewer[6] support the detection of both code smells and some design or architectural smells. These architectural smells have been defined by Marinescu [106]: Cyclic Dependency, SAP breaker and Unstable Dependency.

1 https://www.hello2morrow.com/products/sotograph
2 https://www.hello2morrow.com/products/sonargraph
3 https://structure101.com/
4 https://www.castsoftware.com
5 http://www.intooitus.com/products/infusion
6 http://www.aireviewer.com

Another commercial tool is Designite[7] that detects several architectural smells, defined by Suryanarayana et al. [149], in C# projects.

Other tool prototypes have been proposed, e.g., SCOOP [100], and one from Garcia et al. [51]. SCOOP use a Prolog rules engine to detect architectural smells but it needs a thorough tagging-concern process of the code. The prototype defined by Garcia et al. [51] retrieves the components/connectors from the code which are used for AS detection.

The Hotspot Detector [114] prototype tool detects five architectural smells, called Hotspot Patterns. It computes architectural smells based on the Design Rule Spaces (DRSpaces) [162]. Two of five detectable architectural smells are based on the history evaluation: Unstable Dependency and Implicit Cross Module Dependency. The detector takes as input several files produced by another tool, called Titan [162]. In particular Hotspot Detector takes a file that contain structural dependencies among files, another with the evolutionary coupling information of files and a file with the clustering information of the files. Moreover, Titan takes dependency information output through the Understand reverse engineering tool.

Given this list of tools: AiReviewer, Designite, CAST, Sonargraph and Sotograph are commercial tools and according to my knowledge the other tools are not yet publicly available. Hence, there needed to be a tool for Java which is able to detect architectural smells, so I addressed this problem through the development of Arcan tool, which is one of the main contribution of my thesis. Arcan is available at `http://essere.disco.unimib.it/wiki/arcan` for free.

2.3 CODE SMELLS AND ARCHITECTURAL SMELLS CORRELATIONS

Many works have been done in the literature on code smells by considering their relation and impact on different features such as faults [59, 88], maintainability [44, 71], comprehensibility [28], change frequency [123, 140], change size [123, 124] and maintenance effort [95, 145]. Fewer works are available on architectural smells. Hence in this section we describe some works of the literature on code smells correlations and works that considered both code smells and architectural smells.

2.3.1 *Code smells correlations*

Pietrzak and Walter [132] described several types of inter-smell relations to support more accurate code smell detection and to understand better the effects caused by interactions between smells. They found different kinds of relations among six different code smells by analyzing the Apache Tomcat project.

Arcelli Fontana et al. [7] analyze 74 systems of the Qualitas Corpus, detecting six smells and some relations among smells and possible co-occurrence of smells. They found a high number of relations among God Class and Data

7 `http://www.designite-tools.com/`

Class, among other code smells that tend to go together and a high number of co-occurrence of Brain Method with Dispersed Coupling, and with Message Chains.

Liu et al. [94] propose a detection and resolution sequence for different smells by analyzing some code smells relations given by commonly occurring bad smells. They analyze if it is better to remove first smell `A` than smell `B`, e.g., Large Class versus Feature Envy or versus Primitive Obsession, or Useless Class versus other smells. They considered 9 code smells and identified 15 relations of this kind.

Yamashita et al. [163] studied possible correlations among smells. They incorporated dependency analysis for identifying a wider range of inter-smell relations, and analyzed one industrial and two open source systems. They found the following relations: collocated smells among God Class, Feature Envy, and Intensive Coupling, and coupled smells between Data Class and Feature Envy.

Moreover, some authors provided some code smells classifications or taxonomy that are useful to capture possible relations existing among smells.

Mäntylä et al. [105] categorized all Fowler's code smells except for Incomplete Library Class and Comments smells into five groups: Bloaters, Object Orientation Abusers, Change Preventers, Dispensables, Encapsulators and Couplers. The study outlined the existence of several relations among smells belonging to the same category.

Moha et al. [116] proposed a taxonomy of smells and described some relations among design smells, as Blob and (many) Data Class, or as Blob and (Large Class and Low Cohesion).

Lanza and Marinescu [80] proposed a classification of twelve smells, called "design disharmonies", into 3 categories: Identity, Collaboration, and Classification disharmonies, and they describe the most common correlations between the disharmonies, in a type of diagram called correlation web. However, these correlations have not been empirically validated.

2.3.2 *Code smells and architectural smell correlations*

There is little knowledge, as outlined by Macia [98], about to what extent code anomalies relate to architectural degradation. We report below some works, where sometimes the term code anomalies is used instead of code smells and architectural anomalies for architectural smells.

Macia et al. [101] analyzed code anomaly occurrences in 38 versions of 5 applications using existing detection strategies. The outcome of their evaluation suggests that many of the code anomalies detected were not related to architectural problems. Even worse, over 50% of the anomalies not observed by the employed techniques (false negatives) were found to be correlated with architectural problems.

In another work, Macia et al. [102] studied the relationships between code anomalies and architectural smells in 6 software systems (40 versions). They considered 5 architectural smells and 9 code smells. They empirically found that each architectural problem represented by each AS is often refined by

multiple code anomalies. More than 80% of architectural problems were related to code anomalies. They found *1)* that certain types of code smells, as Long Method or God Class, were consistently related to architectural problems, *2)* that the highest percentages of code smells that introduce architectural problems occur for God Class, Long Method and Inappropriate Intimacy instances, and *3)* that the occurrences of both God Class and Divergent Change smells in the same code element are strong indicators of architectural problems, i.e., Scattered Functionalities that violates the Separation of Concerns design principle. But the study reveals that no type of code smell stands out as the best indicator of harmfulness respect to architecture degradation.

Oizumi et al. [122] propose to study and assess to what extent code smells agglomerations help developers to locate and prioritize design problems. They proposes also to consider not only the structural relations existing among code smells, but also the semantic relations to find more powerful smells agglomerations in order to identify design problems.

Oizumi et al. [121] analyzed 7 systems and suggest that certain topologies of code smells agglomerations are better indicators, than other, of architectural problems. They have considered six code smells detected through the rules of Lanza Marinescu [80] and 7 architectural smells detected through the rules defined Macia [98].

In Chapter 6 I will describe my work and the results obtained on code smells and architectural smells correlation.

2.4 STUDIES ON SOFTWARE QUALITY PREDICTION AND EVOLUTION

2.4.1 *Studies on software quality prediction*

As outlined before, the identification of issues is important, but it could be even more important to predict them before they actually appear. In order to study methods to predict issues, it is necessary an analysis through the development history of projects.

A number of works in the literature use history-based analysis of projects to predict different issues like code smells, changes or bugs, that impact software quality. For example, Palomba et al. [127] defined HIST (Historical Information for Smell deTection), an approach exploiting change history information to detect instances of five different code smells. With respect to bug prediction many works exist, e.g., Khomh et al. [150] explore the presence of antipatterns for bug prediction by analyzing multiple versions of Eclipse and ArgoUML, and Palomba et al. [128] explore if it is possible to improve bug prediction performances using an Intensity Index for code smells [15]. With respect to changes, Code Churn is one of the most used measures when dealing with change prediction, and it is also used, e.g., as a predictor of post-release failures [117].

Romano et al. [134] use antipatterns to predict source code changes, finding that classes affected by antipatterns change more frequently along the evolution of a system. Malhotra and Khanna [104] use search-based techni-

ques and exploit software metrics to predict changes in object-oriented software. Oyetoyan et al. [126] perform an empirical study on different versions of 11 systems to analyze circular dependencies and change proneness and they found that classes involved in circular dependency are more change prone.

Kouroshfar et al. [76] analyzed if co-changes spanning multiple architecture modules are more likely to introduce bugs than co-changes that are within modules. Their results show that the co-changes that cross architectural module boundaries are more correlated with defects than co-changes within modules, implying that, to improve accuracy, bug predictors should also take the software architecture of the system into consideration.

2.4.2 *Studies on software quality evolution*

Different studies in the literature focused on software quality evolution, taking into account the evolution of different quality metrics or the evolution of other issues, such as code smells.

Olbrich et al. [123] investigated two code smells, God Class and Shotgun Surgery, by analyzing the historical data over several years of development of two large scale open source systems. They show that it is possible to identify different phases in the evolution of code smells during the system development and that code smell infected components exhibit a different change behavior: the classes infected with code smells have a higher change frequency.

Vaucher et al. [159] studied in two open-source systems the "life cycle" of the God Class smell: how they arise, how prevalent they are, and whether they remain or they are removed as the systems evolve over time.

Chatzigeorgiou et al. [37], and Peters et al. [131] considered selected properties of code smells, e.g., their evolution and longevity. The authors observed that the number of code smells in software systems increase over time and developers almost never invest significant effort in removing them. This finding was further confirmed by [21], who reported results of a survey aimed at understanding the longevity of code smells. They observed that code smells frequently persist in source code for a long time and developers witheld from refactoring them to avoid API modifications.

In Chapter 6, I will describe the work done on architectural smells prediction and evolution.

2.5 TECHNICAL DEBT INDEXES

Many tools are available, addressing software quality and architecture assessment, e.g., through metrics computations, or code and design anomalies detection, i.e., code smells [48], architectural smells [52, 93] and antipatterns [8]. Detecting these anomalies is useful to identify problems to be solved through the right refactoring steps, but this does not provide us an indication of the overall quality assessment of a project. Hence, some

tools offer some kind of Technical Quality Index, often called, e.g., Technical Debt/Severity, Deficit Index, that offers an evaluation of the overall quality of an analyzed project. In the thesis, the term Technical Debt Index (TDI) is referred to any kind of quality index computed by the tools. These indexes are derived in different ways and take into account different features. The attention is focused on the TDI provided by known tools.

In this section I describe five tools that provide some kind of TDI: CAST, inFusion, Sonargraph, SonarQube and Structure101. The choice of the tools is due, with respect to my knowledge, to the availability of tools providing this kind of Index and my experimentations of all of them on several OSS projects. For example, I experimented, with the aim to refactor or evaluate some architectural problems on existing systems, SonarQube, InFusion and Structure101 on 4 projects [18], Sonargraph and Structure 101 on 2 projects [16], and Sonargraph, SonarQube and inFusion on one project [11].

Obviously, other tools are available, which are able to compute a huge number of metrics, also related to architectural issues, e.g., Massey Architecture Explorer computes an Antipatterns Score [46] and the Tangledness metric [143], Lattix provides Stability, Cyclicality, and Coupling metrics, and STAN supports different R. Martin's metrics [109]. These kinds of metrics are very interesting, but they do not represent (alone) a TDI trying to summarize the overall quality of a system, as those index considered in this thesis.

In in particular I describe how the TDI is computed by the following versions of the tools: CAST 7.3.2, inFusion v.1.8.5, Sonargraph v.8.8.0, SonarQube v.5.2, Structure101 v.4.2.10071. SonarQube is free and open source, while the other ones are commercial tools. The data described were gathered through the use of the tools, their documentation, and by communicating with their technical support, when needed.

2.5.1 *CAST*

CAST[8] defines Technical Debt as the future costs attributable to known structural flaws in production code that need to be fixed, a cost that includes both principal and interest. CAST estimates the amount of principal in the Technical Debt (hereafter called TD-Principal or TDP) of an application based on detectable structural problems. TDP is a function of three variables: *1)* the number of must-fix problems in an application, *2)* the time required to fix each problem, *3)* and the cost for fixing a problem. Each detected problem can affect one or more *Health Factors*: Robustness, Performance Efficiency, Security, Transferability and Changeability. Scores assigned to each internal quality characteristic are aggregated from the component to the application level and reported on a scale from high to low risk, using an algorithm that weights the severity of each violation and its relevance to each Health Factor. CAST assumes that an IT organization would fix 100% of the high severity

8 http://www.castsoftware.com/

problems, 50% medium severity, and no more than 0% of low severity. Weights are assigned to severity levels using the following schema:

- Low severity = Weight of 1, 2 or 3
- Medium severity = Weight of 4, 5 or 6
- High severity = Weight of 7, 8 or 9

To keep the estimate of TDP conservative, the tool assumes that problems can be fixed in 1 hour. CAST sets the labor rate to an average of $75 per hour. The initial formula (and parameters) used to compute TDP is the following:

$$\begin{aligned} TDP = (75\tfrac{\$}{hr}) \times (1hr.) \times \\ ((\Sigma high\_severity\_violations) \times (1) + \\ (\Sigma medium\_severity\_violations) \times (0.5) + \\ (\Sigma low\_severity\_violations) \times (0)) \end{aligned}$$

2.5.2 *inFusion*

inFusion[9] (IF) supports the evaluation of software quality with a focus on code smells (CS) and architectural smells (AS), called *design flaws*, detected by evaluating different metrics. Design flaws are used as the basis for the computation of the *Quality Deficit Index* (QDI), which is reported as a global score or grouped by quality dimension (Complexity, Encapsulation, Coupling, Inheritance, Cohesion). The QDI, the detected design flaws and metric values can be browsed and filtered in different ways, e.g., by quality dimension, by package, or by focusing on single flaws or flawed entities.

The QDI evaluates the impact of all detected design flaws using three factors:

- *Influence* (I_{flaw_type}) This factor expresses how strongly a type of design flaw affects four criteria of good design [40]. It uses a three-level scale (high, medium, low) to characterize the negative influence of a design flaw on each of the four criteria. The assignment of high/medium/low to each design flaw is reported in [108]. I_{flaw_type} is computed as the weighted arithmetic mean of numerical values assigned to the three levels for each of the four criteria.
- *Granularity* (G_{flaw_type}) In general, a flaw that affects methods has a smaller impact on the overall quality than one that affects classes; consequently, it is assigned a weight to each design flaw according to the type of design entities that it affects: class $\rightarrow$ 3 and method $\rightarrow$ 1.
- *Severity* ($S_{flaw\_instance}$) The first two factors refer to design flaw types, which means that all instances of the same flaw weigh equally; since not all cases are equal, for each flaw a severity score is defined, based on its most critical symptoms, measured by one or more metrics. Severity scores span the range 1–10 (low–high).

9 https://www.intooitus.com/, its evolution at http://www.aireviewer.com

Based on the three factors, inFusion computes the Flaw Impact Score (FIS) of a design flaw instance, and derives the QDI value, combining FIS with the size (KLOC) of the system, as follows:

$$FIS_{flaw\_instance} = I_{flaw\_type} \cdot G_{flaw\_type} \cdot S_{flaw\_instance}$$

$$QDI = \sum_{k \in flaw\_instances} FIS_k / KLOC$$

2.5.3 *Sonargraph*

Sonargraph (SG) is meant to support quality controllers, software developers, architects, and consultants. One of its main function related to architectural debt evaluation is the ability to detect deviations from a *defined architecture*, where the developer specifies which dependencies are allowed (or not) between the elements of the system. Moreover, the tool computes different metrics and detects code smells and violations to programming best practices. All detection results can be traced across different versions of the same software.

Sonargraph's Structural Debt [112] is quantified through two measures: *Structural Debt Index* (SDI) and *Structural Debt Cost* (SDC). The first measure is a score computed as the weighted sum of the type dependencies that would need to be cut to break all cyclic package dependencies. The second measure (SDC) is computed by multiplying the SDI by a constant time amount. The tool analyzes the dependency graph to find a good breakup set to disentangle cyclic nodes. The algorithm output is a set of links that need to be removed. The SDI metric is computed by multiplying the number of links to be removed by 10 and then adding the weight (number of dependencies) for each link. As for the SDC metric, the suggested time factor is between 6 and 10 minutes for each SDI point, as a starting point. Then it is possible to define programmers' hour costs, to obtain an estimation of the cost of correcting the issues in the system. Both measures can be computed at the level of system, project or build unit.

2.5.4 *SonarQube*

SonarQube (SQ) is a platform to manage code quality. Its main features are the ability to check large sets of coding rules and to gather software metrics. Examples of rules are the verification of coding constraints and the verification of the range of metric values. Currently, SonarQube's TDI computation does not take into account architectural or dependency information. Each rule is classified in five categories of increasing gravity: Info, Minor, Major, Critical, Blocker. Rule violations are reported as Issues, and can be browsed using different criteria, or shown in source code.

SonarQube implements the *SQALE* [85] model for the estimation of Technical Debt. We experimented with the free plugin integrated in the platform[10]. There are three values computed on the analyzed project, i.e., Technical Debt (TD), Technical Debt Ratio (TDR) and SQALE Rating (SR).

TD represents the time needed to fix all the Issues (i.e., rules violations) detected by SonarQube. A remediation cost (resolution time) is assigned to each issue, and TD is computed as the sum of all the single remediation costs. The TD value associated to single issues can be customized only by purchasing the commercial version of the SQALE plugin. As a result of an aggregation, TD can be associated to any project element (project, sub-project, file) or any *quality characteristic* (e.g., maintainability, security). The TDR is a derived index, obtained as: $TDR = technical\_debt/estimated\_development\_cost$. $estimated\_development\_cost$ represents the estimated time needed to rewrite the project from scratch. In SonarQube, this is set to $LOC * 30minutes$. Please note that in the formula reported above, the two variables are time measures, and need to be in the same unity of measure. The TDR is produced to allow better comparability among different projects. Finally, the SQALE Rating is a letter from A (best) to E (worst), obtained by applying thresholds to the TDR value: A=0–0.1, B=0.11–0.2, C=0.21–0.5, D=0.51–1.

2.5.5 *Structure101*

Structure101[11] (S101) is a tool specialized in architectural evaluation and provides an automatically reconstructed view of the software architecture with different kinds of refactoring support. Structure101 shows a *Structural over-Complexity (SoC)* view to estimate the percentage of the system involved in architectural issues. The view displays two measures: %Tangle and %Fat, i.e., the percentage of metrics Fat and Tangle in the system. The Tangle metric is the count of tangles in packages or classes; specifically, a tangle is a dependencies subgraph containing one or more connected cycles. The Fat metric measures the complexity of the system. At the method level, it is computed using Cyclomatic Complexity (McCabe's metrics), while at any other level it corresponds to the number of inter-dependencies existing among items (e.g., classes, packages). The *Excessive Structural Complexity*, in Structure101, is called *XS* ("excess") and is computed on every item: methods, classes, packages, components. XS represents an estimation of the portion of the LOC of an item that are "affected" by Fat or Tangle.

In Chapter 7, I will describe a proposal of a new indexes, called Architectural Debt Index.

2.6 ARCHITECTURAL SMELL REFACTORING

Different works have considered the impact of code smell [48] refactoring on some quality metrics, e.g., [5, 12, 35], but few works address the problem

10 http://docs.sonarqube.org/display/SONARQUBE52/Technical+Debt
11 http://structure101.com/products/

of the refactoring of architectural smells or other architectural issues. We introduce below some of them.

Bourquin et al. [30] analyze a mid-sized Java enterprise application and show how through the combination of several tools and techniques, they can identify opportunities for high-impact refactorings, which are refactorings with a strong impact on the quality of the system's architecture.

Samarthyam et al. [139] outline that architectural smells and the corresponding refactorings haven't received a large attention from the software engineering community and hence they motivate the need of architecture refactoring and present few potential research directions for architecture refactoring.

Amirat et al. [3] explored the idea of architectural refactoring and introduced an approach for refactoring component based software architecture artifacts using graph transformations. In particular, they used a specific graph transformation tool called AGG (Attribute Graph Grammar) in order to get rid of some architectural bad smells.

Stal [147] analyzed different architecture refactoring scenarios and proposed a catalogue of architecture refactoring steps.

Terra et al. [155] described the preliminary design of a recommendation system to provide refactoring guidelines for developers and maintainers during the task of reversing an architectural erosion process.

Dietrich et al. [46] present a novel approach to detect starting points for architectural refactoring of systems based on the analysis and manipulation of the type dependency graph extracted from programs. They validated their approach by using a set of four antipatterns that are known to compromise modularisation of programs and they found that in most cases, their approach is able to detect high impact refactoring opportunities.

Laval et al. [82] propose eDSM, an enhanced Dependency Structure Matrix (DSM) using colors to distinguish direct and indirect cycles and cells enriched with the nature and strength of the dependencies. They applied eDSM on several systems and assert that their tool allows developers to get a better understanding of cycles between packages, helping to resolve them.

In [16] I described an experience on identifying architectural erosion problems on existing open source software projects through the support of two well known tools, Sonargraph [168] and Structure101 Studio [60]. Moreover, we outline if the tools provide useful hints in repairing some of the detected problems and if the suggested restructuring actions effectively help in the simplification of the system.

In [18] I removed some architectural problems through refactoring steps and we checked the impact that the refactoring has on different quality metrics. In particular, we focused the attention on some Quality Indexes computed by four tools: SA4J, Structure101, Sonarqube and inFusion. These tools have been used also for the detection of the architectural problems.

3

EXPERIENCE REPORTS ON THE DETECTION OF ARCHITECTURAL ISSUES THROUGH DIFFERENT TOOLS

Architectural erosion and degradation [58] of an application lead to architectural debt, and cause problems to the maintainability and evolution of the application. These problems have to be identified and then solved through the right refactoring and reengineering steps. The earlier structural problems are recognized, the easier it is to repair them, but their identification is not an easy task and it can be very expensive, even for experienced software engineers. It is possible to capture signals of architecture erosion in different ways by exploiting several tools with the aim to identify architectural violations, architectural smells [52] or other relevant features.

This chapter describes the analysis done through different tools towards the evaluation of different issues and the impact of their refactoring on quality indexes. Three studies were performed using mainly tools providing technical debt indexes. The studies are part of an incremental study performed to better understand the different features of the tools, their weakness and how they behave in different situations. I outline that the tools I experienced are commercial tools except for SonarQube.

SECTION 3.2: It is provided the experience report in using three known tools to capture information that can be useful to identify and evaluate possible sources architectural debt of an application [11]. I outline the main differences among these tools and the results they produce by analyzing a project.

SECTION 3.3: It is provided the experience report in applying two tools by analyzing 2 projects [16]. It is described the experience in detecting problems that could cause architecture degradation, and in repairing some of these problems directly through the support of the tools, by removing for example cyclic dependencies, or by manually changing the code to remove some architectural violations.

SECTION 3.4: It is provided the experience on repairing and removing possible architectural smells and antipatterns on open source projects [18], by monitoring the impact on 4 TD indexes through different 4 tools.

The tools and their characteristics are reported in the following section.

3.1 TOOLS FOR EVALUATING CODE AND ARCHITECTURAL ISSUES

Several tools have been developed, providing different functionalities that can be used to evaluate architecture degradation. These tools support software analysis in different ways and collect different data. Some tools are proprietary and some are open source. They can detect different kinds of

Table 3.1: Tools feature overview

Tools	Metrics	Code Smells	Architectural Smells	Antipatterns	TDI	Refactoring Suggestions	Warning Violation	Cluster View	Architecture Violation
STAN [120]	44	0	2	1	No	No	Yes	Partial	No
MAE [45]	10	0	1	5	No	No	No	Partial	No
inFusion [67]	61	21	3	0	Yes	No	Yes	No	No
Lattix [81]	9	0	1	0	No	No	Yes	No	No
JDepend [39]	6	0	1	0	No	No	No	No	No
SA4J [64]	5	0	1	6	Partial	No	No	No	No
Understand [141]	47	1	0	0	No	No	Yes	No	No
Bauhaus [133]	12	2	0	0	No	No	No	No	No
IntelliJ IDEA [70]	250	9	0	0	No	Yes	No	No	No
CodePro Analytix [54]	35	2	0	0	No	Yes	Yes	No	No
SonarQube [146]	35	2	1	0	Yes	No	Yes	No	No
Sonargraph [168]	103	15	1	0	Yes	Yes	Yes	No	Yes
Structure101 Studio [60]	6	0	1	0	Partial	Yes	Yes	Yes	Yes

code and design anomalies, i.e. code smells [48], architectural smells [52, 93] and antipatterns [27] . Moreover, they can compute different categories of metrics (e.g., coupling, complexity, cohesion, size) at code and design level. In particular, those regarding dependency or modularity are important indicators of architectural debt [90, 118]. However, it is not easy to fix the right thresholds to identify the critical metric values and to decide what to do to improve the system by exploiting only these values.

In Table 3.1 well known tools are reported which offer different functionalities useful for software quality assessment and for identifying architectural debt.

The following features are considered in Table 3.1:

- *Metrics*: the number of metrics supported by the tool.
- *Code Smells*: the number of detected code smells as defined, e.g., by Fowler [48] or Lanza and Marinescu [80].
- *Architectural Smells*: the number of detected architectural smells [53, 93].
- *AntiPatterns*: the number of detected structural antipatterns [8, 27], e.g., Breakable, Butterfly, Hub; the Tangle antipattern here is excluded, because the same concept is covered by the Cyclic Dependencies architectural smell.
- *Technical Debt Index*: if the tool provides some kind of Technical Debt (TD) or quality index giving an overall evaluation of the analyzed system.
- *Refactoring Suggestions*: if the tool supports refactoring and suggests the way to solve the identified problems.

- *Warning Violations*: if the tool provides some contextual warning of the detected violations.
- *Clustered View*: if the tool provides an automatic clustered view of the system architecture, grouping together tied components and displaying them in a graphical view.
- *Architectural Violations*: if the tool is able to detect some kinds of violations of a specified architecture.

The above features were considered because they can be useful for the analysis, but not all the tools computing metrics, or detecting code smells, or providing the same or similar features, are cited in the table; many other tools exist, but the tools included are providing more than one of the features outlined Table 3.1.

An interesting characteristic of the tools reported in Table 3.1 is that no one of them uses historical information regarding the system to discover architectural issues. Instead, Sonargraph, SonarQube and Structure101 allow recording analysis snapshots and track them over time, to understand the trend in the quality of the analyzed project.

Moreover, we can observe that only inFusion, not more available, detects 3 architectural smells, while the other tools do not recognise architectural smells or only one.

3.2 TOOL SUPPORT FOR EVALUATING ARCHITECTURAL DEBT

In this study [11], the attention is particularly focused on evaluating architectural debt by identifying possible architectural violations, dependency violations or other kinds of structural violations. In this perspective, the aim is to explore the usefulness of a set of tools in identifying architectural anomalies and possibly in providing some recommendations or refactoring guidelines for developers and maintainers during the task of reversing an architectural erosion process. A first way to face and get perhaps some more concrete information on the quality of an application could be through an Index, such as Technical Debt Index, which aggregates the data collected through metrics and the different architectural violations, providing us an indication of the quality and architectural debt of the application.

In Table 3.1, it is shown that Sonargraph, SonarQube and inFusion tools provide a Technical Debt Index estimation.

It is not easy to define a formal framework for the comparison of this kind of tools, because the information they provide can be very different, at different levels of granularity and often named in different ways. Hence, the attention is focused on comparing some architectural evaluation issues:

- the results inspection capabilities of the data the tools provide;
- the data extracted by the tools on the same system.

The three tools, SonarQube v.5.1, inFusion Hydrogen v.1.8.5, and Sonargraph-Quality v.7.2.2, are experimented by analyzing an application, RepoFinder [9] made of 417 Java files composed of 53,640 text lines, which I have developed for the crawling and analysis of FLOSS projects; the main aspects of the overall experimentation are discussed, comparing the considered architectural evaluation issues provided by the three tools.

3.2.1 *Evaluating the results inspection of the tools*

In this section, the experimentation is described through the three tools by analyzing RepoFinder.

3.2.1.1 *Evaluation through Sonargraph*

One of the main functions of Sonargraph is related to architectural debt evaluation through the ability to detect deviations from a *defined architecture*, where the developer specifies which static dependencies are allowed (or not) between the elements of the system. Another relevant function is the detection of Cyclic Dependency among compilation units, Java packages, or higher architectural aggregates. Moreover, the tool computes different metrics and detects violations to programming best practices. The only supported architectural smells are *Cyclic Dependency*. Detection results can be traced across different versions of the same software, showing the trends existing in the available measures.

The logical software architecture for a software system can be specified in two different dimensions: *horizontal*, defining layers identifying technical components, and *vertical*, defining slices that identify the structure of the domain. The specification supports the definition of the allowed or forbidden dependencies among the abstraction levels. In this way, the tool can check if the specified architecture is respected by the analyzed system.

It is also possible to simulate the application of *refactoring* on the model of the system that is created by Sonargraph, and check the consequences at the structural level before any effort is spent.

The Quality Index of Sonargraph's *Structural Debt* [112] is quantified by two measures: Structural Debt Index and Cost, as described in Section 2.5.

Results inspection: The Exploration tab (see Figure 3.1) shows allowed/disallowed links between project elements. Elements, e.g., layers, packages, types, are reported in a list with drill-down capabilities, so that it is possible to inspect the project at different abstraction levels at the same time. Allowed links are shown in green and disallowed links in red. In particular, links on the right side of the list represent dependencies creating cycles between components, while the ones on the left side of the list represent all the other dependencies. The detected cycles can be inspected in a different dedicated view, as the extracted metrics and their violations.

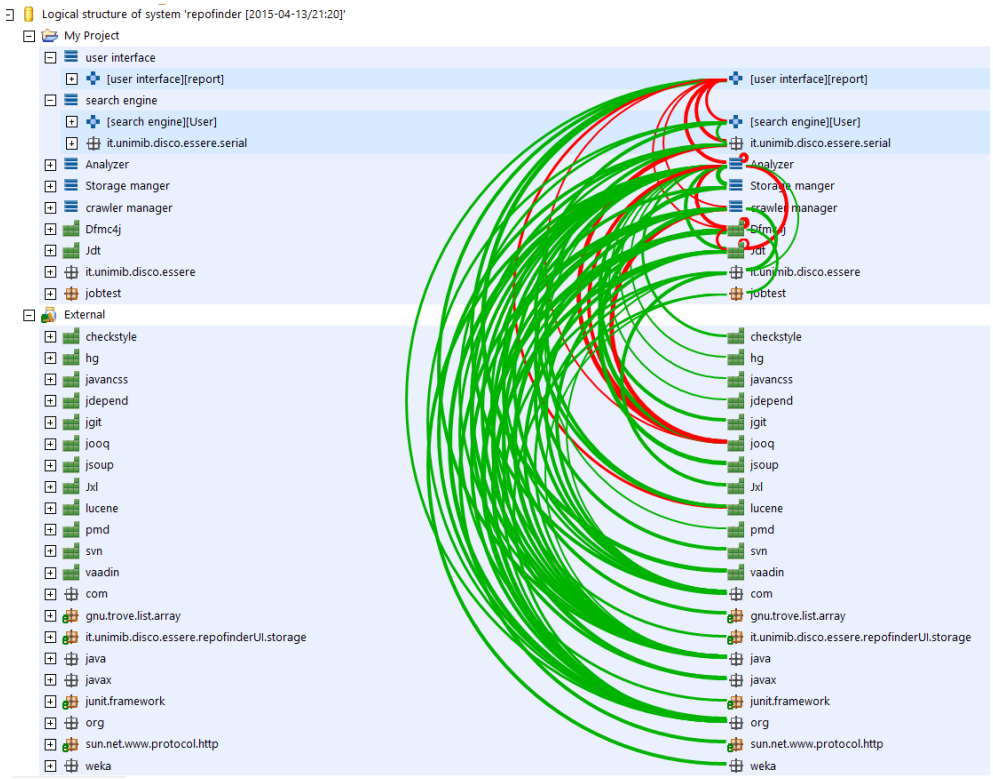


Figure 3.1: Sonargraph - package exploration view

3.2.1.2 *Evaluation through SonarQube*

SonarQube is a platform to manage code quality. Its main features are the ability to check large sets of coding and design rules and to gather software metrics. Examples of rules are the verification of coding constraints, both generic or specific to a particular technology, and the verification of the range of metric values. Each rule is classified in five categories of increasing gravity. Violations of the rules are reported as "*Issues*" in the platform, and can be inspected using different criteria, or shown in source code.

Among its different functions, SonarQube deals with dependencies. It computes the dependency graph of the analyzed system, and provides a dependency matrix through its "*Design*" view. Therefore, it supports the detection of the *Cyclic Dependency* architectural smell. The tool provides the number of cycles, the number of dependencies to cut at the directory or file level, and a *Directory Tangle Index*, i.e., the ratio between the number of dependencies to cut to remove cycles and the total number of dependencies.

SonarQube's implements the *SQALE* [85] model for the estimation of technical debt, we applied the free Technical Debt[1] capability integrated in SonarQube. There are basically three values computed on the project, i.e., TD, Technical Debt Ratio and SQALE rating, as described in Section 2.5.

Results inspection: The inspection of the dependency matrix view allows two operations on a cell: highlighting the name of the Components relative

1 http://docs.sonarqube.org/display/SONAR/Technical+Debt

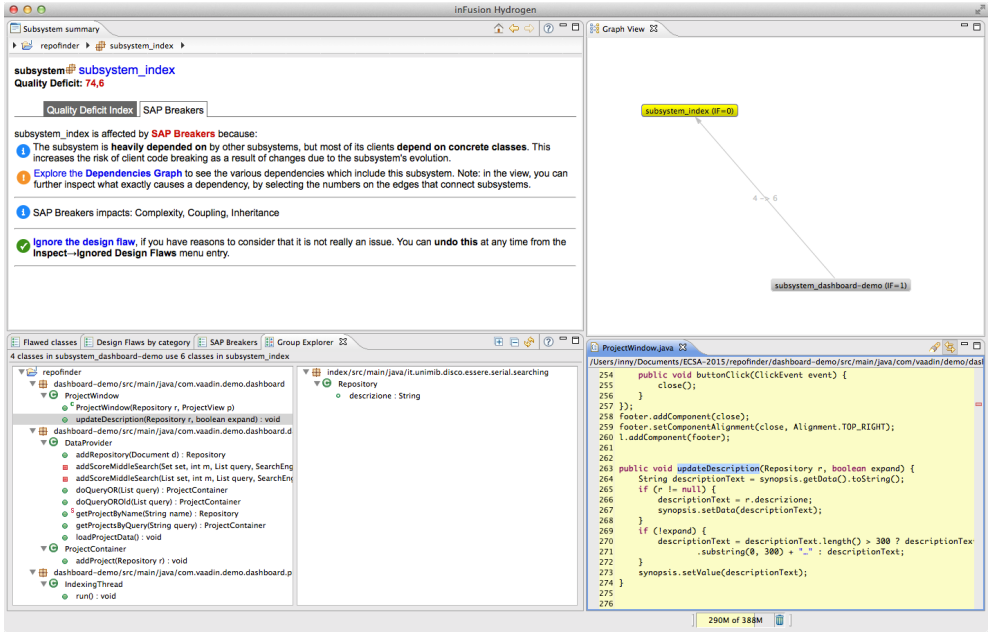


Figure 3.2: inFusion - SAP Breaker instance inspection

to the cell and opening the list of elements on both ends of the dependency represented by the cell. The matrix uses a color code: components highlighted in green depend on the ones in orange (see Figure 3.4). When inspecting a dependency's contents there is no way to be directed to the specific point(s) in the source code where the dependency exists.

3.2.1.3 *Evaluation through inFusion*

inFusion supports the evaluation of software quality with a focus on code smells, called Design Flaws (i.e. code smells and architectural smells). Design Flaws are detected by evaluating different metrics, which are also available to the user and reported in different views. Design Flaws are used as the basis for the computation of the *Quality Deficit Index (QDI)* [108] of inFusion, which is reported as a positive value aggregating the detected Design Flaws: as a global score or also grouped by quality dimension (Complexity, Encapsulation, Coupling, Inheritance, Cohesion), as described in Section 2.5.

Architectural smells in inFusion are categorized as Design Flaws [80], and are all associated to the Coupling quality dimension, plus other dimensions for each specific case. In addition to the *Cyclic Dependency* architectural smell, supported also by the other two tools, inFusion supports other two architectural smells (described in Section 2.2): *SAP Breaker* and *Unstable Dependency*.

Results inspection: The inspection is focused now on architectural smells. These smells are defined over different evaluations of dependencies among modules, and they can be inspected through a graphical representation of the slice of the dependency graph regarding the specific Flaw. From this

view, it is possible to access the source code associated to each class or method.

The tool allows to show, in a single view (see Figure 3.2), the description (upper left), graph (upper right), code (lower right) and dependency details (lower left) of an architectural smell instance.

3.2.1.4 *Concluding Remarks on Results Inspection*

In inFusion, the report of Cycle Dependency have duplicated records, in fact, are reported once for each module containing it, and not as distinct entities. Selecting a cyclic dependency, it is possible to see both the corresponding dependency graph and the respective source code. As it is explained above, SonarQube represents cycles on the dependency matrix. This representation is very clear and unambiguous, but does not scale well on a large number of elements. Moreover, it is not possible to see dependency cycles with more than two elements, and the inspection function does not allow to jump into the code where the problem is located. Sonargraph provides a dedicated view for cycles (see Figure 3.3), at different levels of granularity. This view provides functions to manage and inspect the found cycles. It also integrates with the architecture specification, by representing forbidden dependencies as red arrows (allowed dependencies are green). An additional function of this view is the computation of "*break-up*" dependencies sets, to suggest possible ways to resolve the detected cycles, associated to different metrics estimating the effort of the resolution.

Summarizing, Sonargraph surely addresses the management of Cyclic Dependency with the larger feature set, while SonarQube provides the poorer experience from this point of view.

3.2.2 *Evaluating the extracted data by the tools*

The architecturally-relevant data extracted by each tool are discussed highlighting differences and similarities. All the tools were applied on the analyzed project two times, one by defining the static *logical architecture* of the project, with the level of detail allowed by the tool, and the other without defining the architecture, and considering packages as the modules of the project.

3.2.2.1 *Evaluation through Sonagraph*

The Exploration view was used (see Figure 3.1) to find cycles and violations of the logical architecture. Some logical architecture violations were discovered; for instance, an access to an external library (JOOQ) was found that should be used only by the storage management layer. The user interface was uses directly some libraries instead of using the correct module. Cyclic Dependency are shown to exemplify how a package is used by the other packages. For example, Figure 3.3 shows in the Cycles tab that the `dfmc4j` package contains a dependency cycle composed of 9 sub-packages. The Cycles tab allows to see the graph of the detected cycles, where nodes may be types, packages or subsystems and (directed) edges represent dependencies.

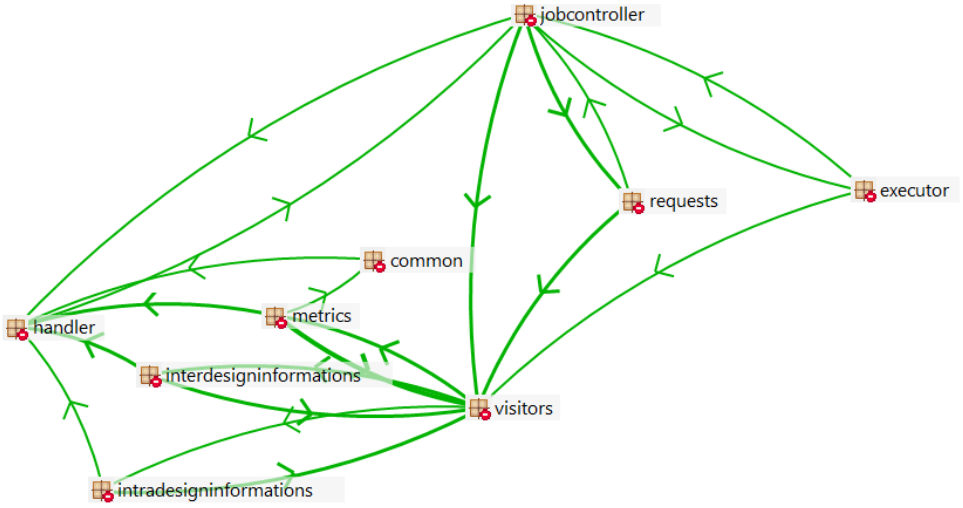


Figure 3.3: Sonargraph - Cycles view

Table 3.2: Sonargraph results overview

	Logical Architecture	
Rule	No	Yes
Structural Debt index	12,563	25,968
Cyclic elements	95	95
Cyclic packages	31	31
Biggest pkg cycle group	9	9
Logical pkg cycle groups	9	9
Layer cycle groups	0	0
Layer group cycle groups	0	0
Architectural Violations	374	1,073

Table 3.2 are reported the main global measures related to architectural evaluation. The definition of the logical architecture influences a lot the Structural Debt index and the number of detected architectural violations. Defining the logical architecture makes the detection of violations more precise, and more of them are detected. This raises the value of the Index, as it is strongly based on the detected violations.

3.2.2.2 *Evaluation through SonarQube*

SonarQube does not analyze dependencies between modules for non-maven projects. This issue is clear by looking at the module dependency matrix (see Figure 3.4) computed on the project (modules have been specified using SonarQube Runner). The alternative way of analyzing dependencies in SonarQube is to avoid the specification of modules and obtain a single dependency matrix. However, due to the number of packages contained in the project, the matrix is large and consequently difficult to read. Another relevant li-

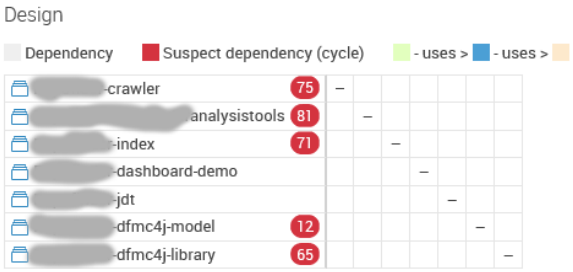


Figure 3.4: SonarQube - dependency matrix among non-maven modules

Table 3.3: SonarQube results overview

	Logical Architecture	
Rule	No	Yes
SQALE Rating	A	A
Technical Debt Ratio	5.80%	5.30%
Technical Debt	238d	216d
Dependencies To Cut (Directories)	20	20
Dependencies To Cut (Files)	304	304
Directory Tangle Index	25.4%	29.4%
Cycles	>25	>25

mitation is that dependencies from external libraries are not shown. This functionality is also available when running SonarQube on Maven projects. Table 3.3 shows the global quality measures of the same project in the two setups. Technical Debt and Technical Debt Ratio are slightly higher without the definition of modules, while the SQALE Rating remains unchanged.

3.2.2.3 *Evaluation through inFusion*

The differences in the two result sets are remarkable. When setting modules manually, the tool found only 3 instances of the SAP Breakers Flaw, while in the other setting it found 14 SAP Breakers and 6 Cyclic Dependency. This difference results in QDI values of 10.7 and 18.3, respectively. Given that the QDI is computed relying on the detected Design Flaws, it is shown how the module setting affects its value.

3.2.2.4 *Concluding Remarks on extracted data*

Some differences were experienced in the two result sets. For example, it was noticed that in Sonargraph the Structural Debt Index rises when the logical architecture is defined. In SonarQube, instead, the SQALE rating is stable and the Technical Debt Ratio is lower when modules are specified. inFusion behaves similarly: its QDI index is lower when modules are specified.

With respect to the detected Cyclic Dependency, Sonargraph and SonarQube detected the same number of cycles in both configurations. inFusion,

instead, computes Cyclic Dependency only among modules; when modules have been specified w.r.t. the logical architecture, no cycles were detected, while when modules have been associated to packages, 3 distinct cycles were detected. inFusion detected the smallest set of Cyclic Dependency; the same cycles have been detected also by the other two tools was checked. Sonargraph and SonarQube found more than 25 cycles. According to SonarQube, 20 folder Cycle Dependency need to be cut, while for Sonargraph there are 9 dependency cycles. It is attributed this difference to the fact that SonarQube shows (and perhaps computes) cycles on pairs of packages, while Sonargraph considers also larger sets. It is verified that the Cyclic Dependency reported by the tools were correct. While inspecting the reported cycles were not resolved them, since they were not harmful in my opinion, and their resolution was not clear. Architectural smells, like code smells, point to *possible* issues, that need to be verified.

3.3 DETECTING AND REPAIRING SOFTWARE ARCHITECTURE EROSION

In this section, we describe the experience on the support provided by tools towards the identification of architectural erosion problems [144] and their solution. In particular, well known product families for software quality assessment are experimented, i.e., Sonargraph [168] and Structure101 Studio [60]. Other tools are available, but Sonargraph was chosen since it has been recognised as the best tool in the previous work [11] (described in Section 3.2) and Structure101, since it is specialized in architectural evaluation and re-engineering support, and takes a different approach with respect to Sonargraph in both the analysis and re-engineering phases. Structure101 is also the only considered tool that provides an automatically reconstructed view of the architecture that clusters related components, with the aim of simplifying the comprehension of the project.

In this study, the architecture of two projects is analyzed, and their re-engineering is attempted according to the tools' suggestions. The experimentation is performed by running the tools on a FLOSS (Free, Libre and Open Source System) project (Ant v1.8.4) and an application developed by me for the crawling and analysis of FLOSS projects, called RepoFinder [9]. By analyzing a project that I have developed, I was able to better capture the functionalities of the tools in reconstructing the software architecture when a reference architecture is available, in better identifying the problems and applying the right suggestions to remove them. Hence, it is described the experience in detecting problems that could cause architecture degradation, and in repairing some of these problems directly through the support of Sonargraph and Structure101, by removing for example Cyclic Dependency, or by manually changing the code to remove some architectural violations.

Few research works in the literature report experiences to control architecture erosion by evaluating the support provided by different tools. As an example, Gorton and Zhu [55] experiment five tools (Understand for Java, JDepend, SA4J, ARMIN, Enterprise Architect) on an industrial Java project composed of 50kLOC. After reconstructing the architecture with each tool,

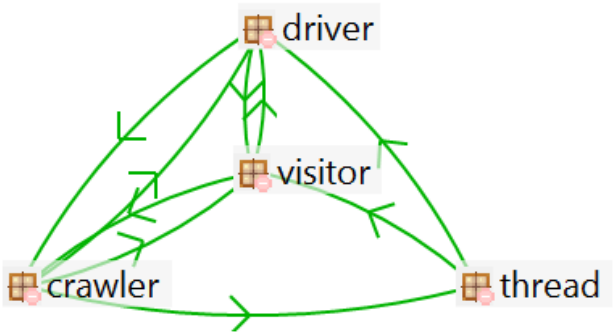


Figure 3.5: Sonargraph - Example of cycle to cut in the Cycles View on RepoFinder

they identify different general and specific points for the improvement of the tools. In this study, two state-of-the-art and complete tools are experienced, and it is identified how they complement each other and the possible improvement directions.

3.3.1 *Detecting and Repairing Design Erosion with Sonargraph*

The experience in detecting and repairing some architectural problems with Sonargraph on the two systems, Repofinder and Ant, is described in this section. The size of the two systems can be found in Table 3.4. A description about some of the principal features and functionalities offered by Sonargraph respect to the study aims is given below. The evolution of these features is also evaluated (see Table 3.4), as indicators of possible architecture erosion, before and after the refactoring steps.

Sonargraph checks the conformance of the software architecture with respect to a predefined model and finds possible violations in the project. The logical architecture can be defined using horizontal layers and vertical slices, each containing some code artifact, as described in Section 2.5. It is also important to define external libraries and subsystems, to better understand the project and how it relates to the external components. Packages and classes have to be assigned to layers, slices, subsystems, libraries and external components. This assignment is realized by specifying wildcard/glob patterns to match the names of packages and classes.

Sometimes, internal components are made for common usages like "*database manager*" (DBM), where the database must be accessed by a single component and the other components have to use it to access the database. In order to verify automatically if this practice is respected, it is necessary to create "*allowed dependencies*" from the DBM to the database driver and from other components to the DBM. Otherwise it is necessary to define "*denied dependencies*" from all components except for DBM to the database. After the definition of the allowed dependencies, defining denied dependencies is redundant.

Sonargraph provides a concise dependency graph called Exploration view, which shows the allowed, denied and Cyclic Dependency, between compo-

Table 3.4: Sonargraph - Extracted metrics

	RepoFinder		Ant	
Name Property	before	after	before	after
Structural Debt Index	25,698	11,080	4,959	4,895
Architecture Violations	1.073	437	2	0
Cyclic Packages	31	22	30	25
Biggest Package Cycle Group	9	9	26	25
Code Smells	5,265	2,902	5,338	5,324
Code Smells/LOC	0.08	0.05	0.05	0.05
Packages	116	124	70	70
Classes	912	819	1,278	1,278
Lines of Code	67,536	58,300	105,007	105,054

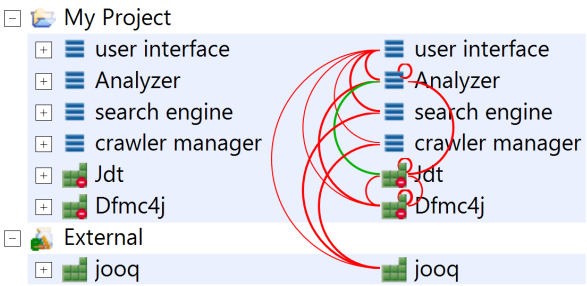


Figure 3.6: Sonargraph - Exploration view on RepoFinder

nents, external components (e.g., libraries) and packages. A Cyclic package group is shown in the Cycles view (see Figure 3.5), where packages are nodes and the edges represent dependencies.

The information extracted from the source code and the architecture is synthesized in a single measure, called Structural Debt Index. It has the aim of being roughly proportional to the effort needed to clean up structural quality problems in the code, as described in Section 2.5.

Detecting Design Erosion

RepoFinder is a project that I designed and developed. For this reason, I were able to fully specify its reference architecture in Sonargraph. Table 3.5 reports the rules used to group packages in components in RepoFinder. "*" means that every class and interface directly contained in the package is considered. "**" considers every subpackage and contained classes or interfaces.

Several architectural violations were found in RepoFinder (1073), as shown in Table 3.4. As it was observed and then checked, the most frequent architectural violations were related to the database usage. The only component allowed to access the database library (JOOQ[2]) is the database manager, but the accesses found from other components, e.g., directly reading tables

2 https://www.jooq.org/

Table 3.5: RepoFinder components definition patterns

Package	Component
it.unimib.disco.essere.serial.**	search engine
it.unimib.disco.essere.serial.*	search engine
it.unimib.disco.essere.serial	search engine
it.unimib.disco.essere.analysis.**	analyzer
it.unimib.disco.essere.analysis.*	analyzer
it.unimib.disco.essere.analysis	analyzer
it.unimib.disco.essere.repofinderUI.tool.**	analyzer
it.unimib.disco.essere.repofinderUI.tool.*	analyzer
it.unimib.disco.essere.repofinderUI.tool	analyzer
it.unimib.disco.essere.dashboard.**	user interface
it.unimib.disco.essere.dashboard.*	user interface
it.unimib.disco.essere.dashboard	user interface
it.unimib.disco.essere.repofinderUI.storage.**	db manager
it.unimib.disco.essere.repofinderUI.storage.*	db manager
it.unimib.disco.essere.repofinderUI.storage	db manager
org.apache.lucene.**	apache
org.apache.lucene.*	apache
org.apache.lucene	apache
org.jooq.**	jooq
org.jooq.*	jooq
org.jooq	jooq
org.jsoup.**	jsoup
org.jsoup.*	jsoup
org.jsoup	jsoup
net.sourceforge.pmd.**	pmd
net.sourceforge.pmd.*	pmd
net.sourceforge.pmd	pmd
jdepend.xmlui.**	jdepend
jdepend.xmlui.*	jdepend
jdepend.xmlui	jdepend
com.puppycrawl.tools.checkstyle.**	checkstyle
com.puppycrawl.tools.checkstyle.*	checkstyle
com.puppycrawl.tools.checkstyle	checkstyle
com.aragost.javahg.**	hg
com.aragost.javahg.*	hg
com.aragost.javahg	hg
com.vaadin.**	vaadin
com.vaadin.*	vaadin
com.vaadin	vaadin
org.tmatesoft.svn.core.**	svn
org.tmatesoft.svn.core.*	svn
org.tmatesoft.svn.core	svn
org.eclipse.jgit.**	git
org.eclipse.jgit.*	git
org.eclipse.jgit	git
javancss.**	javancss
javancss.*	javancss
javancss	javancss
jxl.**	jxl
jxl.*	jxl
jxl	jxl

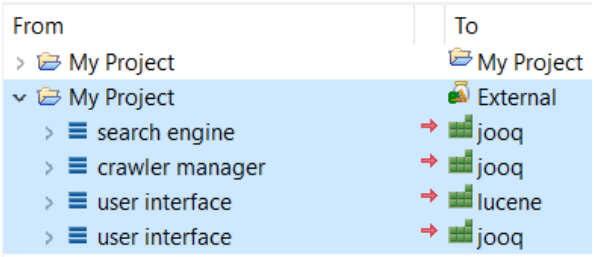


Figure 3.7: Sonargraph - Architecture Violations in RepoFinder

through JOOQ. The second most frequent architecture violation is related to the direct access to the Apache Lucene index. This, in fact, has to be accessed only by the crawling and indexing components of RepoFinder.

A large number of smells was also found. For what concern code smells, Sonargraph identifies few of the ones defined by Fowler and other code anomalies of different kinds and granularity, as unused/missed/hideable-something. In particular, 4987 code smells were found, with a high number of unused methods (2985), hideable public methods (600), missed abstraction methods (522), unused classes unreachable (452).

Moreover, the number of Cyclic Packages detected in RepoFinder and the biggest package cycle, which is composed by 9 packages are outlined in Table 3.4. Cyclic packages, also called Cyclic Dependency, represent one of the most critical architectural smell [53, 93].

Sonargraph requires the definition of the reference architecture of the project, to provide the detection of architectural violations. For Ant, architectural documentation is very scarce. A reference to the separation of packages in modules was found in the Javadoc contained in the project's binaries[3]. In the documentation, allowed and disallowed dependencies are not defined. The only information available is the division of packages in modules.

The packages of Apache Ant 1.8.4 have been grouped in five components: core, task, types, util and tools. Table 3.6 summarizes the rules used to group packages.

Sonargraph reported only two architectural violations in Ant (see Table 3.4), where two classes located in sub-packages of `org.apache.tools.ant` access classes outside the `ant` package, e.g., classes contained in `org.apache.tools.zip`. Compared to RepoFinder, Ant has lower Structural Debt Index but contains a much larger package cycle group (26 packages). The number of identified code smells in the two systems is comparable (5338 vs 5265), while code smell density is higher in RepoFinder.

Repairing Design Erosion

In order to highlight architectural violations, Sonargraph provides the "Exploration View", which shows dependencies using colored edges: green for correct dependencies and red for violated dependencies. Figure 3.6 shows the view over RepoFinder.

3 http://archive.apache.org/dist/ant/binaries/apache-ant-1.8.4-bin.zip

Table 3.6: Ant components definition patterns

Package	Component
org.apache.tools.ant	Core
org.apache.tools.ant.*	Core
org.apache.tools.ant.taskdefs	Task
org.apache.tools.ant.taskdefs.*	Task
org.apache.tools.ant.taskdefs.**	Task
org.apache.tools.ant.taskdefs.optional	Task
org.apache.tools.ant.taskdefs.optional.*	Task
org.apache.tools.ant.taskdefs.optional.**	Task
org.apache.tools.ant.types	Types
org.apache.tools.ant.types.*	Types
org.apache.tools.ant.types.**	Types
org.apache.tools.ant.types.optional	Types
org.apache.tools.ant.types.optional.*	Types
org.apache.tools.ant.types.optional.**	Types
org.apache.tools.ant.util	Util
org.apache.tools.ant.util.*	Util
org.apache.tools.ant.util.**	Util
org.apache.tools.bzip2	Tools
org.apache.tools.bzip2.*	Tools
org.apache.tools.bzip2.**	Tools
org.apache.tools.mail	Tools
org.apache.tools.mail.*	Tools
org.apache.tools.mail.**	Tools
org.apache.tools.tar	Tools
org.apache.tools.tar.*	Tools
org.apache.tools.tar.**	Tools
org.apache.tools.zip	Tools
org.apache.tools.zip.*	Tools
org.apache.tools.zip.**	Tools

In RepoFinder, the indications about violated dependencies reported on the use of database libraries were followed and removed the violated dependency. In the logical architecture of the project, the JOOQ library was defined as an external sub-system. The Architecture Violations view lists the existing violations (see Figure 3.7). The information of this view was used as starting point to perform the refactoring. The refactoring operations can be summarized with three refactoring actions: *(i)* the first is to create an external component dedicated to the use and management of the database, *(ii)* the second one is to eliminate, where it is possible, all usages and references to the JOOQ subsystem from all the components, and *(iii)* the third one is to use the new database manager component to access and use the JOOQ library. These actions significantly lowered the Structural Debt Index of the project, by more than a half.

In Ant, the two reported architectural violations were solved. In order to solve these violations, the methods from the called class were moved to the calling class, removing the dependency between the packages.

Maintainability and Evolution using SonarGraph

In Table 3.4, the evolution of the data collected before and after repairing some of the identified problems were reported for the two systems.

In RepoFinder, the number of Architecture Violations and Cyclic Packages and the Structural Debt Index decreased significantly. The three values are tied since the Index is computed over the other two measures. While code smells are not directly related to the applied refactoring, their number changed drastically: unused methods lowered by a half (5265 $\rightarrow$ 2860), hideable public methods by a half (600 $\rightarrow$ 301), missed abstraction methods by a tenth (522 $\rightarrow$ 66); unused classes unreachable did not lower so drastically (452 $\rightarrow$ 398). Lines of Code significantly decreased, too.

In Ant, refactoring actions slightly decreased the Structural Debt Index. The removal of Cyclic Dependency was attempted, exploiting the recommended dependency cuts proposed by Sonargraph. However, after solving five cycles, it was clear that this operation was not effective in tackling the measured Structural Debt. The number of code smells decreased by only 14 code smells, of these types: unused attributes, unused methods.

Considering the refactoring actions executed on the two systems, as far as it is observable from the data the Structural Debt Index has been proportional to the effort spent. The reason why no more changes were applied to Ant is mainly that the tool did not provide many feasible refactoring opportunities. Some relevant proposed changes were extremely complex, and were not addressed. The tool is certainly useful both for detecting and repairing architecture defects and monitor the quality of an application during maintenance. However, on a project where the architecture is not known in much detail, the proposed refactoring opportunities can be not effective or very difficult to address.

3.3.2 *Detecting and Repairing Design Erosion with Structure101*

In this section, the same analysis performed with Sonargraph by exploiting Structure101 Studio for Java v4.2.10071 (Structure101 in the following) [60] was executed. The tool provides different views, with the ability to define, inspect and manipulate the model of the analyzed project, to show the collaboration between code items, understand the class hierarchy, navigate the call method graph and detect Fat items and/or Tangles.

Fat corresponds to complexity measured on a single item, at any level of granularity. Cyclomatic Complexity (CC) is used to measure Fat on methods, while at any other level of granularity Fat is measured as the number of inter-dependencies existing among the item's contents. For example, Fat in packages is the number of dependencies among the contained classes. Size measures are not used to measure Fat, since it is assumed that the complexity of the control flow or of the structure makes an item hard to understand, and not, e.g., its number of sequential lines of code. Items are considered "Fat" when their Fat value exceeds a defined threshold. For example, the default threshold for methods is 15.

Tangles are cyclic, package-level dependencies. As widely known, they drive up the coupling of a structure, making it much harder to understand, extend or modify. Moreover, activities such as regression testing, software reuse, and porting to modern frameworks become virtually impossible. Structure101 organizes Tangles into levels by calculating the Minimum Feedback Set (MFS), i.e., a minimum set of dependencies that, if removed, would render the dependency graph acyclic, and by showing the levels as if the MFS dependencies were removed. Tangles are computed at the package and design granularity levels. Each package or component is assigned to a Tangle percentage, reflecting the portion of its content participating to a tangle.

By using Fat, Tangle, and a Size metric (LOC), Structure101 computes another metric called XS ("excess"), on every item: methods, classes, packages, components. XS represents an estimation of the portion of the LOC of an item that are "affected" by Fat or Tangle (see Section 2.5).

Detecting Design Erosion

Structure101 was used to analyze the software architecture of Ant 1.8.4 and RepoFinder. Structure101 reconstructs the software architecture and shows the top list of *Tangles* and *Fat* elements. The structural over-complexity graph, shown in Figure 3.8 [4], shows the percentage of the elements of the project are considered "Tangled" or "Fat". The graph classifies a project as structured or unstructured. If the percentage of Tangled and Fat items in the project is near to 0%, the project is structured, while if it is near to 100% the project is considered unstructured. The graph shows how much structure the project has, and how much it will have after the planned refactoring actions.

4 the background color is not relevant to understand the figure

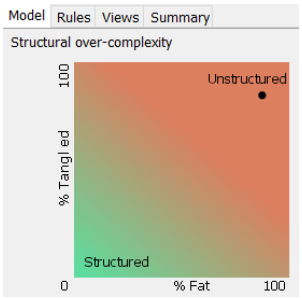


Figure 3.8: Structure101 - Structural over-complexity view on Ant

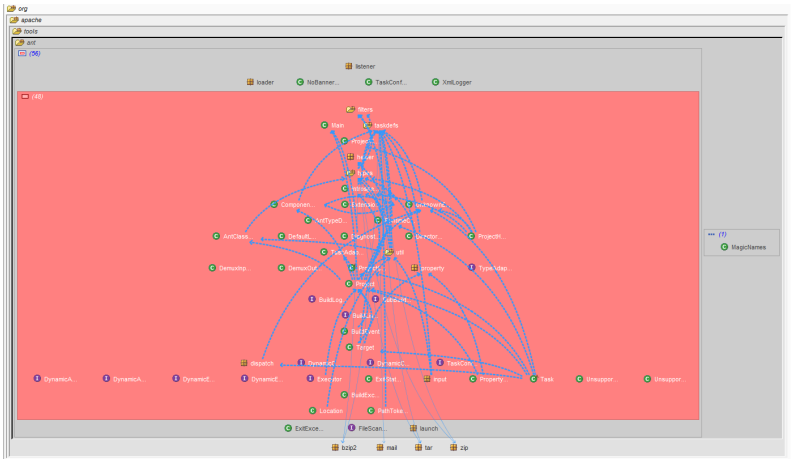


Figure 3.9: Structure101 - Exploration View on Ant

The reconstructed architecture is shown using the Levelized Structure Map (LSM), which allows choosing different grouping methods: cohesive cluster, tangles and tagged items. Figure 3.9 shows Ant's LSM, applying the Cohesive Cluster grouping method. System items are grouped according to their common dependencies and LSM levels. It helps to understand Fat regions, and can help in the process of subdividing a Fat item in several smaller items. The large red/dark rectangular group depicted in Figure 3.9 is a tangle of 48 items, contained in a clustered group of 56 items belonging to the `org.apache.tools.ant` package. There is also an orphan item (`MagicNames` class) that is not referred by items of its package. Items can be tagged, simplifying their comprehension process, by allowing to easily locate them in different views and to understand their collaboration with the rest of the project.

Defining the logical architecture of the project is important to check possible architecture violations. This feature is provided by Structure101 in the rule tab shown in Figure 3.10. It is possible to define more than one diagram and check different configurations of the project architecture, counting how many class dependencies violations there are w.r.t. the designed one. The diagram defines layering through the top-down arrangement of cells. In general, cells may only be used by others in higher layers, but it is possible to define exceptions and to override this rule, by drawing green (continuous line) arrows (to allow dependencies) and red (dashed line) arrows (to disal-

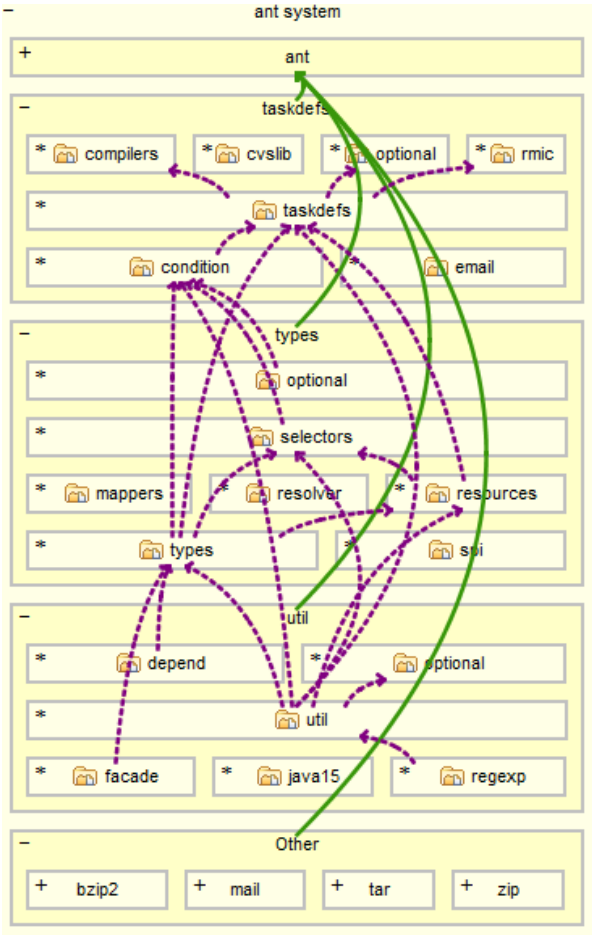


Figure 3.10: Structure101 - Architecture Diagram of Ant

low dependencies). Black arrows are reported only when there is a violation among the designed components.

Other views exist for the detailed evaluation of Tangles or Fat items, to focus attention on tangles and their elements, to show the collaboration among different source items, and display an overview of the hierarchical composition of code and call graphs.

Structure101 found 3 tangles at package level, and 26 at class level in Ant (see Table 3.7). It also found 29 class-to-class dependency violations from the defined architecture.

RepoFinder (see Table 3.7) contains more tangles than Ant, but they are smaller: the largest package tangle in RepoFinder contains 9 classes, while in Ant the largest tangle contains 26 classes. The result is consistent with the one reported by Sonargraph and described in Section 3.3.1.

Tables 3.8-3.9 show the density of Tangle and Fat items in the project, and their respective thresholds, at all granularity levels. In Ant, 9 of 17 defined component are tangled. Tangles are the prominent XS violation for Ant. In fact, Tangles cover 63% all the XS measured on the proect. The density of Fat items is generally low (1%–3%).

Table 3.7: Structure101 - Detected Tangles

		Ant		RepoFinder	
Level		before	after	before	after
Package	**#Items**	70	70	308	289
Package	**#Tangles**	3	2	9	7
Package	**#Tangled items**	30	28	30	25
Package	**Biggest**	26	26	9	9
Class	**#Items**	845	850	1,467	1,288
Class	**#Tangles**	26	14	39	13
Class	**#Tangled items**	300	272	296	200
Class	**Biggest**	181	181	74	74

Table 3.8: Density of tangled and fat item in Ant

		Threshold	before	after
Tangled (design)	**#Offenders**	0	9 of 17	8 of 17
Tangled (design)	**%Offenders**	0	53%	47%
Tangled (design)	**%XS**	0	63%	63%
Fat (design)	**#Offenders**	120	0 of 17	0 of 17
Fat (design)	**%Offenders**	120	0%	0%
Fat (design)	**%XS**	120	0%	0%
Fat (package)	**#Offenders**	120	2 of 70	2 of 70
Fat (package)	**%Offenders**	120	3%	3%
Fat (package)	**%XS**	120	11%	12%
Fat (class)	**#Offenders**	120	16 of 1,270	16 of 1,270
Fat (class)	**%Offenders**	120	1%	1%
Fat (class)	**%XS**	120	13%	14%
Fat (method)	**#Offenders**	15	95 of 11,205	95 of 11,205
Fat (method)	**%Offenders**	15	1%	1%
Fat (method)	**%XS**	15	12%	12%

Table 3.9: Density of tangled and fat item in RepoFinder

		Threshold	before	after
Tangled (design)	**#Offenders**	0	16 of 202	14 of 190
Tangled (design)	**%Offenders**	0	8%	7%
Tangled (design)	**%XS**	0	69%	66%
Fat (design)	**#Offenders**	120	0 of 202	0 of 190
Fat (design)	**%Offenders**	120	0%	0%
Fat (design)	**%XS**	120	0%	0%
Fat (package)	**#Offenders**	120	0 of 308	0 of 289
Fat (package)	**%Offenders**	120	0%	0%
Fat (package)	**%XS**	120	0%	0%
Fat (class)	**#Offenders**	120	17 of 1,816	16 of 1,614
Fat (class)	**%Offenders**	120	1%	1%
Fat (class)	**%XS**	120	17%	19%
Fat (method)	**#Offenders**	15	41 of 12,212	38 of 9,500
Fat (method)	**%Offenders**	15	0%	0%
Fat (method)	**%XS**	15	14%	15%

In RepoFinder, the density of Tangles is lower, but they still represent the largest part of the XS in the project. Fat items are present only at class level and method level, with density value of 0%–1%.

Repairing Design Erosion

Structure101 builds a model of the analyzed project, containing, e.g., its structure, architectural violations, XS metric values and offenders. Each model is stored in a repository that can be accessed by different developers. A single repository can contain the models of several versions of several different systems.

To support the removal of tangles from source code, Structure101 provides an action plan, which is also stored in the model, containing the recommended refactoring to perform on the project, to apply the refactoring actions simulated through the tool. Every single action is an atomic source code change that involves one or two items (e.g., add class `A` or move method `m()` from `A` to `B`).

The action plan can be accessed from the Eclipse IDE, but cannot be automatically applied. Developers must choose how and if to follow the action plan. The selection of an item in the action plan redirects to the file interested by the action. Structure101 does not provide a way to apply a refactoring automatically. Since Eclipse provides some refactoring automation, e.g., extract class with parameters, move method, it is possible to exploit them, but

they are not triggered by the Structure101 plugin. Moreover, the action plan is not synchronized with the IDE. To update the action plan, a full project analysis must be executed.

In Ant, 12 tangles at class level and 1 at package level were resolved, removing dependencies in the Minimum Feedback Set, to reduce the refactoring effort. The resolution of tangles did not effectively enhance the structure of the project measured by the tool, probably due to the fact that only relatively small tangles were resolvable, and their resolution covered a small part of the project. While working on Ant, which is a project not know, especially about its design, Structure101 was particularly suitable to understand the project without prior architecture knowledge. The tool provides a clear view of the project, and allows experimenting with different refactoring simulations through its interactive user interface.

In RepoFinder, the problem of database access was noticed also, that was spread across the project components. Refactoring actions were applied to move to the database component a lot of dependencies regarding the JOOQ library, similarly to what has been done with Sonargraph. The tool provided a precise list of the refactoring actions, but the overall effort has been the same was spent with Sonargraph, because there is no automation of refactoring actions, and the additional information regarding clustering sets is confusing.

Maintainability and Evolution using Structure101

The measures reported by Structure101 on Ant (see Table 3.8) did not change significantly after the application of the refactoring actions. Most measures did not change their value at all, with the exception of the percentage of Tangle at design granularity. The number of Tangles has been decreased by refactoring actions performed at both class level and package level. Despite the resolution of 12 class Tangles and 2 package Tangles, the improvement in the structural measures has been very low.

The refactoring actions applied to RepoFinder lowered significantly its size, as shown in Table 3.9: number of methods ($12,212 \rightarrow 9,500$), classes ($1816 \rightarrow 1614$) and package ($308 \rightarrow 289$). This reduction of the project was not proportionally followed by a reduction of Fat items, leading to a slight increase in the Fat percentage. This aspect is something probably worth noting, since Size does not influence Fat metrics, but a 22% change in the size of the project probably has an impact on its understandability. However, Size is always shown in the tool together with the other metrics, allowing the user to take both indicators into consideration.

3.3.3 *Discussion and Lessons Learned*

According to the experimentation of Sonargraph and Structure101, was observed that:

- The automatic clustering features provided by Structure101 are particularly useful to understand and analyze unknown systems. The de-

tection of Tangles (or Cyclic Dependency) is another feature relevant for this task, and is supported by both tools.

- The architecture definition functionalities of Sonargraph allow a more precise architecture specification. The effect of the specification, i.e., the assignment of items to components, and the detected violations, can be inspected interactively.
- Action plans generated by Sonargraph contain high-level restructuring tasks, e.g., cutting the dependency between two classes. Structure101 gives the developer more support, providing a detailed action list that realizes the high-level action plan.
- The tools do not associate refactoring actions to time/cost/effort measures. Moreover, the structural measures provided by the tools are not explicitly related to time or cost, and refactoring actions are not associated to the expected gain in the respective structural measures. This results in a lack of prioritization of the generated action plans.
- The action list provided by Structure101 is more detailed than the action plan generated by Sonargraph, and can be inspected in the IDE. Despite these advantages, there is space for improvement. A first issue to solve is the lack of interaction with the IDE: when a refactoring step coming from the action list is applied, it is not removed from the action list. On large action lists, this can lead to confusion and lack of manageability. A second issue is the fact that the action list items are not contextualized with the project. The selection of an item can lead to the relevant file, but not to the precise point of intervention, slowing down the application of refactoring.

Conducting this experimentation on detecting and repairing software architecture problems gave considerable insights into various facets of the experimented tools and how they tackle the essential problems that must be dealt with during an evaluation and reconstruction effort. In the insights below is outlined what in my opinion need still to be addressed:

1. as I observed, the experimented tools, and those cited in Table 3.1, do not support software architecture history analysis. This could be a relevant direction of improvement of the existing tools. Some techniques exist that exploit this historical information, e.g., the detection of evolutionary coupling can be used to discover unseen dependencies.
2. The experimented tools do not allow to execute the planned refactoring actions automatically or semi-automatically. This would be of great value for developers, and a gap to bridge for tool vendors. In fact, most IDEs support refactoring actions, and a contextualized action plan could easily trigger them in the IDE, leaving to the developer only the effort of deciding details like the name of the new classes or methods to be created.

3. Tools do not address the dynamic analysis of the project yet. This aspect of dynamic analysis is surely more difficult to tackle, but has the potential of revealing very important information about the connections among the components of the project, especially in distributed or service-oriented architectures.

4. The only architectural smell detected by the two tools is Cyclic Dependency (Tangle). Many other architectural smells have been defined [93, 114], addressing very different properties of the software architecture.

3.4 THE IMPACT EVALUATION OF ARCHITECTURAL PROBLEMS REFACTORING

In this study the attention is focused on the detection of some architectural problems, as architectural smells [52] (AS) and antipatterns [33] (AP), and on the evaluation of several metrics through the support of different tools. In particular the study aims to evaluate the impact that the refactoring of some AS or AP has on the values of some metrics. Among them, particular attention is given to Quality Indexes evaluated by some tools, as Quality Deficit or Technical Debt Indexes. For this reason,four tools were experimented (inFusion, SA4J, Sonarqube and Structure101), that are able to compute some kind of Index providing an evaluation of a project, as Table 3.1 shows. Structure101 was preferred to Sonargraph since in [16] we argued that Structure101 provides better support in the case of unknown architecture. The experimentation was done on four OSS projects taken from the Qualitas Corpus (QC) [154].

Refactoring is expensive and it would be useful to focus the attention on the refactoring of the most critical problems. Architectural problems as AS are considered more critical, for example, respect to code smells and if they are not removed, it could be possible to assist to a progressively software architecture erosion and degradation [102]. Hence, in this study the investigation is started from the impact of the refactoring of these *more critical problems* on Quality Indexes. A developer could be interested to know the benefits of refactoring this kind of problems on some quality features, as those captured by the indexes. Moreover, he could be interested to know which kinds of problems the tools are able to detect, problems that for the developer could be difficult to be manually identified.

The goal of these study is to better understand if the refactoring of AS or AP is easy or not, if the refactoring of these problems improves or not the quality of a system according to the different Quality Indexes, and if the definition and computation of the considered Quality Indexes could be improved.

3.4.1 *Study Setup*

Analyzed Systems

Four systems written in Java were analyzed, reported in Table 3.10. The selected systems belongs to two main categories from Qualitas Corpus: Middleware and Tool.

Table 3.10: Systems analyzed and refactored

	Middleware		**Tool**	
	Quartz	**Informa**	**Jag**	**Commons**
Version	1.8.6	0.7.0	6.1	3.2.1
LOC	40477	12830	20385	42285
# of methods	2659	1297	1434	3694
# of classes	244	160	257	428
# of packages	25	13	16	12

Quartz and Informa are projects of the Middleware category. Quartz is an open source job scheduling library that can be integrated within virtually any Java application. Informa is an open source project that provides a news aggregation library based Java platform.

JAG and Commons-collections are projects of the Tool category. Jag is an application that creates working J2EE applications. Commons-collections is a standard for collection handling in Java, e.g, Maps, Bidirectional Maps, Bags.

Tools

Four tools were used to assess software quality from different points of view and to detect different architectural problems. These tools were used since they are able to detect AS or AP and compute different metrics and Quality Indexes, as shown in Table 3.1: SonarQube (SQ), inFusion (InF) and Structure101 (S101); Structural Analysis for Java (SA4J)[5], developed by IBM, computes the dependencies in a project, provides several views and the detection of several antipatterns.

Data Collection: Architectural smells and antipatterns

The AS reported by inFusion are the following three (see Section 2.5): Cyclic Dependency, Unstable Dependency and Stable Abstraction Breaker.

The AP detected by SA4J are the following:

- *Tangle* - is a large group of objects or package whose relationships are so interconnected that a change in any one of them could affect all the

5 IBM Alphaworks - http://alphaworks.ibm.com

others. Large Tangles are a major cause of instability in large systems (Tangles exist both class and package level).

- *Local Butterfly* - is an object with many immediate dependents. It has many immediate relationships when another object depends upon it.
- *Global Butterfly* - is an object with many global dependents. Many objects are affected when a Global Butterfly changes.
- *Local Breakable* - is an object that has many immediate dependencies and relationships when it depends on another object. Local Breakables are typically undesirable because they "know too much".
- *Global Breakable* - is an object that has many global dependencies and is often affected when any other object is changed in the system. They have to be avoided, as they indicate fragility and lack of modularity in the system.
- *Local Hub* - is an object with both many immediate dependencies and dependents. It has both many immediate relationships that affect other objects and where other objects affect it.
- *Global Hub* - is an object with both many global dependencies and many global dependents. It is often affected when any other object is changed, and it affects a significant percentage of the system when it changes.

The Cyclic Dependency smell detected by InF corresponds to the Tangle at package level detected by SA4J and S101. Table 3.12 shows the number of these AS and AP detected on all the four systems. Moreover, the number of Tangle are reported in the Table (S101 and SA4J) and Cycles (SQ) at class level and the number of the different AP (SA4J) always at class level.

Collected Data: Quality Indexes

As outlined before, the above tools were selected because they are all able to compute a Quality Index. In particular the tools compute the following Quality or Technical indexes (see Section 2.5):

- *Software Quality Assessment based on Lifecycle Expectations* [87] (SQALE) and *Technical Debt* (TD) - SQALE and TD are computed by SonarQube. There are basically three values computed on a project, i.e., TD (day), TD Ratio and SQALE rating.
- *Quality Deficit Index* (QDI) - It is computed by inFusion, through the number of code smells (CS) and architectural smells (AS) found in a system.
- *Stability Index* - It is computed by SA4J and measures the stability of a system, where systems are considered stable when internal and external dependencies are balanced.

- *Structural over-Complexity* (SoC) - It is computed by Structure101 (see Figure 3.11).

Refactoring

The same refactoring process was applied to remove AS and AP. The systems were analyzed with all the tools every time the refactoring was performed, in order to check if the Quality Indexes were improved or not by following few steps:

1. Search and remove Tangles. For the Tangles at package level, S101 suggestions were followed to solve them. For the Tangles at class level, the javadoc documentation of the systems was read, and the dependencies were reassigned, working directly on the source code;
2. Search and remove AS of InF. If the Cyclic Dependency or Tangles were already removed, then for other AS was checked, as SAP Breakers and Unstable Dependency. Then, the code was refactored in order to remove them;
3. Remove Fat items found by S101. Fat at different granularity was removed using different techniques: on packages (extract class refactoring), on classes (extract super class, extract subclass refactorings) and on methods (extract method refactoring).

Some of the refactoring suggestions of S101 were exploited, Eclipse IDE (v.4.5 Mars) was used to apply them and then the refactoring in the code was manually checked. In some cases, as for example for long Extract Method, the refactoring was done manually and not through Eclipse.

The refactoring steps have been applied by four Master students in computer science, following the course of Software Evolution and Reverse Engineering at University of Milano Bicocca, with a main focus on software quality assessment through code and architectural smells detection and refactoring. The same students applied the four tools to detect the different architectural problems, to performe the different refactoring steps to remove them and to evaluate the different Indexes before and after the refactoring. They all worked together on the four analyzed systems, each one checked every step and they discussed the critical cases on the refactoring or not of a problem respect to another one in order to improve quality indexes. They performed the task in 3 weeks part-time. At the end, the results were checked by a PhD student.

3.4.2 *Results*

The obtained results for each system are described in Tables 3.11, where some metrics and all the indexes computed by the tools are shown and in Tables 3.12 the number of AS and AP, before and after refactoring.

In the following subsections the attention is focused on each system and the impact of the refactoring steps applied to remove AS and AP is evaluated.

Table 3.11: Metrics and Indexes measured for every system

			Middleware						Tool					
			Quartz			Informa			Jag			Commons		
Tool	**Name**	**T**	**O**	**R**	**D**	**O**	**R**	**D**	**O**	**R**	**D**	**O**	**R**	**D**
Structure 101	Tangle score (XS) (Package)	I	6436	0	-6436	672	0	-672	27386	0	-27386	9496	0	-9496
	Fat (Package)	M	0	0	0	0	0	0	0	1	1	0	0	0
	Fat (Class)	M	5	2	-3	0	0	0	3	3	0	6	6	0
	Fat (Method)	M	20	20	0	7	7	0	17	17	0	4	4	0
	Fat score (XS) (Package)	I	0	0	0	0	0	0	0	13314	13314	0	0	0
	Fat score (XS) (Class)	I	12098	5683	-6415	0	0	0	4166	4166	0	3304	3331	27
	Fat score (XS) (Method)	I	5088	4348	-740	1997	1997	0	2258	2258	0	103	103	0
inFusion	QDI	I	11.80	11.2	-0.6	6.4	6	-0.4	12.8	10.9	-1.9	2.5	0.9	-1.6
	CC (Average Method)	M	1.88	1.85	-0.03	1.65	1.65	0	2.17	2.15	-0.02	1.74	1.75	0.01
	Coupling Deficit Index	I	19.5	15.3	-4.2	8.8	7.0	-1.8	23.4	16.2	-7.2	6.4	0.8	-5.6
SonarQube	Tangle Directory Index (%)	I	3.8	0	-3.8	5.4	0	-5.4	48.3	0	-48.3	50.6	0	-50.6
	TD (Day)	I	61	61	0	30	31	1	55	55	0	52	52	0
	TD Ratio	I	4	4	0	5.1	5.2	0.1	5.8	5.8	0	3.3	3.4	0.1
	SQALE	I	A	A	–	A	A	–	A	A	–	A	A	–
	Number of issues	M	2277	2296	19	1022	1028	6	2910	2923	13	1875	1884	9
	LOC	M	24539	24684	145	9722	9754	32	15375	15421	46	25069	25016	-53
	Duplications (%)	M	3.8	3.3	-0.5	5.5	5.5	0	1.6	1.6	0	7.6	7.8	0.2
	Dependencies to cut (Pkg)	M	7	0	-7	4	0	-4	16	0	-16	12	0	-12
	Dependencies to cut (Cls)	M	9	0	-9	9	0	-9	65	0	-65	129	0	-129
SA4J	Stability Index (%)	I	90	90	0	85	86	1	85	85	0	97	97	0
	Number of Object	M	195	211	16	160	160	0	142	143	1	279	278	-1
	Number of Packages	M	28	35	7	17	17	0	23	20	-3	15	14	-1
	Number of Relationships	M	1209	1275	66	1013	1004	-9	994	976	-18	1334	1323	-11

Columns: T (Type), O (Original version), R (Refactored version), D (Differences);
Type: M (Metric), I (Index); **Granularity:** Cls (Class), Pkg (Package)

Table 3.12: Architectural smells and Antipatterns detected on every system

			Middleware						Tool					
			Quartz			Informa			Jag			Commons		
Tool	**Name**	**Granularity**	**O**	**R**	**D**	**O**	**R**	**D**	**O**	**R**	**D**	**O**	**R**	**D**
S101	Tangle	Package	1	0	-1	2	0	-2	1	0	-1	1	0	-1
S101	Tangle	Class	4	2	-2	3	1	-2	2	2	0	2	0	-2
InF	Cyclic Dependencies	Package	11	0	-11	4	0	-4	12	0	-12	10	0	-10
InF	Unstable Dependencies	Package	1	0	-1	0	0	0	0	0	0	2	0	-2
InF	SAP Breakers	Package	1	0	-1	0	0	0	0	0	0	2	0	-2
SQ	Cycles	Class	17	0	-17	5	0	-5	41	0	-41	35	0	-35
SA4J	Tangle	Package	1	0	-1	1	0	-1	1	0	-1	1	0	-1
SA4J	Tangle	Class	4	2	-2	3	1	-2	2	2	0	2	0	-2
SA4J	Breakable (Local)	Class	19	20	1	28	27	-1	28	28	0	17	17	0
SA4J	Breakable (Global)	Class	31	32	1	96	70	-26	51	51	0	0	0	0
SA4J	Hub (Local)	Class	8	8	0	4	4	0	14	14	0	0	0	0
SA4J	Hub (Global)	Class	1	1	0	7	6	-1	33	33	0	0	0	0
SA4J	Butterfly (Local)	Class	24	28	4	19	19	0	30	29	-1	24	24	0
SA4J	Butterfly (Global)	Class	20	22	2	27	25	-2	67	70	3	5	5	0

Columns: O (Original version), R (Refactored version), D (Differences)
Tools: S101 (Structure 101), InF (InFusion), SA4J (SA4J), SQ (Sonarqube)

Results for Quarz

After the refactoring, the following architectural problems were removed:

- 60% of Tangles recognized by SA4J and Structure 101. The percentage of Tangle Directory Index computed by SQ is decreased to zero;
- 60% of class fatness (Fat score) computed by S101, removing 3 fat class;
- 100% of AS recognized by InF, which corresponds to the 20% of Design Flaws (AS and CS) and to the 18% of Entity Flaws recognized by InF.

The QDI of InF is improved, from 11.8 to 11.2 (5%) and in particular the Coupling Deficit Index. The Technical Debt Ratio of SQ is still constant after the refactoring. Duplication percentage is decreased from the original to the refactored version. Some compromises were accepted during the refactoring operations:

- The removal of Tangles led to the introduction of some new AP, like Local and Global Butterfly and/or Local and Global Breakable: 2 Breakable and 6 Butterfly AP were introduced. Hub is constant to 9.
- The Number of Relationships of SA4J increased due to the introduction of 16 new objects.

Results for Informa

The following architectural problems were removed after the refactoring:

- 80% of Tangles recognized by SA4J and S101. The percentage of Tangled Directory Index detected by SQ is decreased to zero.
- 100% of AS recognized by InF, which corresponds to the 10% of Design Flaws and to the 10% of Entity Flaws recognized by InF.

27 Breakable, 1 Hub and 2 Butterfly AP were removed by refactoring the above AS and AP, . The InF QDI was improved from 6.4 to 6.0 (7%) and was slightly increased the Stability Index of SA4J. The Technical Debt Ratio of SQ slightly increases after the refactoring steps.

Results for Jag

The following architectural problems were removed:

- 33% of Tangles recognized by SA4J and S101. the percentage of Tangle Directory Index computed by SQ was decreased to zero;
- 100% of AS recognized by InF, which corresponds to the 22% of Design Flaws and to the 15% of Entity Flaws of InF.

1 Local Butterfly was removed while Breakable and Hub AP remain constant at zero by removing the above AS and AP.

The QDI of InF was improved from 12.8 to 10.9 (15%) and in particular the Coupling Deficit Index. The Technical Debt Ratio of SQ is constant at 5.8 and the Number of Relationships of SA4J decreases.

The following compromises were accepted during the refactoring operations:

- Some new AP, like Global Butterfly, were introduced by removing the Tangles.
- A Fat element at package level was introduced with an associated Fat score of 13,314, because this was the only way to remove Tangles and improve other indexes. Fat issues are tied to complexity measures, and by reducing coupling, it could increase complexity, as shown in this case. Different strategies to remove Tangles were tried, but an increment of the Fat score was always observed.

Results for Commons Collection

The following architectural problems were removed:

- 100% of Tangles recognized by SA4J and Structure 101. The percentage of Tangled Directory Index computed by SQ were decreased to zero;
- 100% of AS recognized by InF, which corresponds to the 35% of Design Flaws and to the 30% of Entity Flaws recognized by InF.

The QDI of InF was improved, from 2.5 to 0.9 (64%) and in particular the Coupling Deficit Index. The AP Breakable, Hub and Butterfly AP are constant to zero. The Technical Debt Ratio slightly increased.

3.4.3 *Observations on the results*

The quality of the four systems was generally improved through the refactoring of the different architectural problems, but the following observations respect to the Quality Indexes and the removed problems are reported.

The SQALE Index is constant at the best value *A* for all the systems before and after refactoring. Technical Debt Ratio is related to the SQALE index and it is constant for the systems, except for Informa and Commons-collection systems where it slightly increases.

The Quality Deficit Index (QDI) is improved (decreased) for every system after the refactoring, this is due to the decreasing value of the design flaws. All the AS detected by InF were removed, resulting in a significant lower value of QDI, except for the Quartz system with only a slightly decrement of the index. A great attention was paid to avoid the introduction of new code smells or other problems During the AS refactoring.

The Stability Index of SA4J remains constant for all the systems. Informa is the only system with a slightly increment of the Stability Index.

The SoC Index of S101 shown in Figure 3.11 has been always improved (decreased near to zero). Jag is the worse case where the tangleness is decreased to zero against an increment of the fatness, but the refactored version of Jag has a lower SoC index. The attention was focused in removing first of all the AS, and after their refactoring a reduction of all the Fat issues was not observed (XS at Package level), probably for the reasons outlined above (lower coupling with an increment of complexity).

Moreover, to highlighting that the tools can compute the same metric by providing different values. This has been observed in the literature also for the very simple LOC metric. In our case for example, the cyclomatic complexity computed by InF that slightly decreases or remains constant, and Fat issues, related to complexity, computed by SoC that behave in the same way, with the exception, as already observed for the Jag system.

The Tangles of SA4J and S101 at package level and all the Cycles detected by SQ have been removed from each system. The Tangles detected by SA4J are the same of the Tangles of S101. However, SA4J did not find one Tangle at package level in Informa found by S101. Finally, the number of Dependencies to cut (SQ), the Tangle Directory Index (SQ) and the Tangle score (S101) decreased to zero.

The number of Hub AP is more or less constant in every system. Generally, the refactoring of all the AS had a small impact on Hub AP, but a greater impact on other AP as Breakable and Butterfly.

3.4.4 *Threats of Validity*

Some limitations of our experimentation are known. The experimentation using four representative tools has been done, but other tools could be considered in future experimentation, as for example Sonargraph and CAST. Moreover, many different architectural problems were removed, but not all of them, as outlined before, for different reasons and compromises.

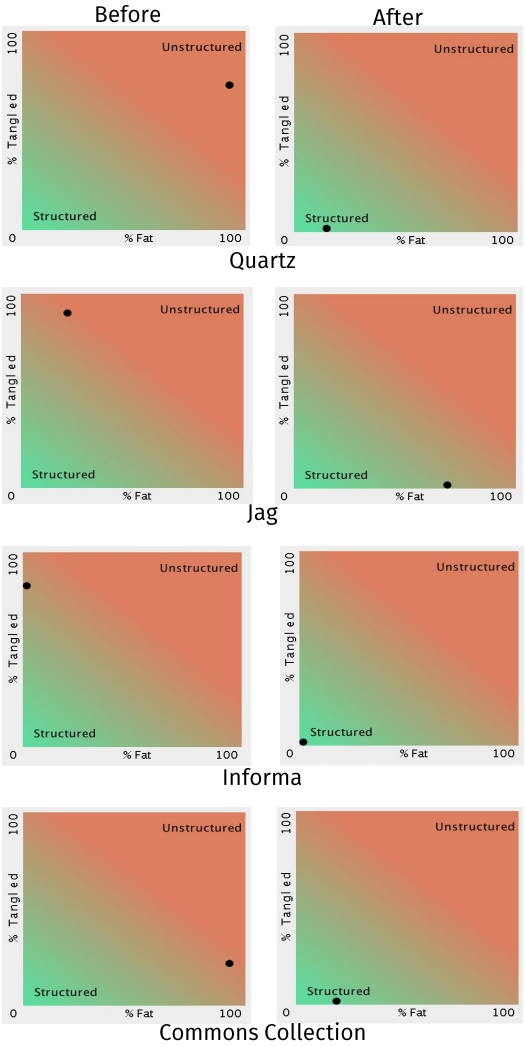


Figure 3.11: S101 - Structural over-Complexity

As outlined in Section 3.4.1, the refactoring steps have been applied by four Master students in computer science, hence also if they had experience on this task, they could have not always made the best choice on the refactoring to be applied or on the problem to be removed.

Only four systems were analyzed, hence, the experimentation has to be extended, but in any case the analysis done on the four systems could be representative of the impact of the refactoring of architectural problems and can give hints on the different problems and compromises to be faced.

In following, the conclusions of this chapter are given about the three studies explained before.

3.5 CONCLUSIONS

Three studies [11, 16, 18] were performed using mainly the tools providing technical debt indexes. In the following, I describe the main conclusions of each study.

3.5.1 *First Study of Section 3.2*

In Section 3.2, the experience in using three tools (SonarGraph, SonarQube and inFusion) able to provide a quality/debt index is described with the aim of evaluating the architectural debt and the overall quality of a software project. With this aim, the answer to the following research question is provided:

RQ1 *Which tool has the largest set of useful function for evaluating architectural debt?*

Even if all the applied tools provide functions to evaluate the architecture, they have different origins, and were initially developed with a different set of core functions. SonarQube is mainly a code checker, and provides lots of low-level checking rules, tracing them on different project versions; it computes Technical Debt taking into account the violations to the rules. inFusion's main focus is on code smells (design flaws), and its QDI measure relies on them and their severity; Sonargraph, instead, is more focused on the evaluation of the architecture, and provides more facilities for this kind of task; the index it provides, in fact, is mainly derived from architectural smells or violations. Hence, Sonargraph provides the largest set of useful functions for the considered task. In the other tools, some key features are missing. For example, SonarQube does not allow to inspect the found dependency cycles; inFusion is better, but has some localized issues, e.g., it reports duplicated instances of dependency cycles, creating confusion in the analysis phase; moreover, it reports less dependency cycles than the other tools. As for the most useful functions provided by the applied tools, I think that Sonargraph's Exploration view allows to quickly inspect most architectural issues from a single graphical entry point, substituting both the dependency matrix and the graph representation of dependencies provided by the other

tools. Moreover, only Sonargraph is able to support refactoring by simulating the removal of dependencies. The other tools allow only to see the issues, but not the way to mitigate them or the effect of the intervention.

I think that this experience report can be useful to developers or maintainers to have a quick indication on the support given by the three tools in identifying architectural debt, but also to tools' developers to improve their tools. For example, the available Indexes summarizing the quality or debt of the projects are not directly useful when evaluating a single project. These measures cannot be interpreted with the aim to understand the overall quality of the analyzed project on a global scale. In other words, the number cannot tell if the project is good or not. The SQALE rating, which should go in this direction, in my experience is overly optimistic. In fact, most of (if not all) the projects analyzed received an A rating, even when significant issues existed. Of consequence, I think that these Indexes are currently useful only on a relative scale, in the case a single team evaluates an entire portfolio of applications. In this case, the Index can be used, e.g., to rank new projects with the respect to the old/existing ones. In this context, the support given by Sonargraph and SonarQube to the management of different versions of the same project, is useful to trace the trend of the Index and other metrics, and to understand if the changes applied between two versions have been harmful for the quality of the system.

3.5.2 *Second Study of Section 3.3*

In Section 3.3, it is described the experience using Sonargraph and Structure101 to detect code and architecture problems and in repairing some of them. The aim of the study was to answer the following research questions:

RQ1 *How the tools differ in supporting the reconstruction of systems where a reference architecture is available or not (known/unknown system)?*

Both tools allow specifying the reference architecture of the analyzed project. Sonargraph was more mature and richer in terms of offered functionalities. Its way of specifying the reference architecture of the system is more detailed, allowing, e.g., to specify layers and slices, and to interactively inspect the packages and classes contained in the specified components. Structure101 handles the component definition mainly relying on the existing packages of the system, which can only be grouped graphically. Wildcard patterns can be used to exclude unwanted items from the architecture by name, but not to define components as in Sonargraph. Components can be organized in a hierarchy, but there are no concepts similar to Sonargraph's layers and slices.

As for the support for architecture evaluation and discovery, without the specification of the reference architecture, the two tools have different characteristics. Structure101 allowed us to discover more relevant architectural issues on Ant, even without the definition of its architecture. In particular,

the Cohesive Cluster grouping method used in the LSM view is particularly effective in highlighting different architectural aggregates, without any prior knowledge. For example, it allowed to discover an architectural violation regarding database access, without the need of defining the reference architecture of the system. Sonargraph, instead, provides the detection of Cyclic Dependency, but for any other evaluation it fully depends on the definition of the reference architecture of the system. For these reasons, it was argued that Structure101 provides better support in the case of unknown architecture.

RQ2 *Do the suggested restructuring actions effectively help in the simplification of the system?*

In both the tools, the reconstructing actions chosen effectively helped in the simplification of the system, in proportion to the needed effort. Both tools do not provide suggestions regarding the priority of the actions to take. Sonargraph allows the engineer to assign a level of priority to each selected refactoring action. The action plan produced by Structure101 is ordered according to the sequence of operations selected in the tool. Sonargraph sorts the action plan items first by priority and then according to the sequence of operations selected in the tool. In both tools, there is no automatic suggestion of the most effective actions to undertake. Moreover, there is no way to trigger automatic refactoring of the code starting from an action plan.

RQ3 *Do the provided Technical Debt measures reflect the effort spent in restructuring the system?*

In both the tools, the provided measures, i.e., the Structural Debt Index and Tangle%/Fat%, in Sonargraph and Structure101 respectively, reflected the effort spent during the refactoring actions on the two systems. The two measures have different scales, but they are proportionally tied to the structural problems of the analyzed systems. However, in both tools it is not possible to reliably estimate the effort needed to execute the action plans.

3.5.3 *Third Study of Section 3.4*

In Section 3.4, our experimentation is described on the removal of some architectural problems, as those represented by AS and AP. Then, the impact of the refactoring of these problems on four systems has been evaluated on some quality issues through the computation of metrics and Quality Indexes.

The aim of the study was to answer the following research questions:

RQ1 *Which is the impact of the refactoring of architectural problems, as architectural smells or antipatterns, on different Quality Indexes?*

Some of the Indexes provided by the tools through the refactoring steps, as described before, in particular QDI and SoC Indexes were improved by removing Tangles of S101, SA4J and SQ were improved by removing all the

AS of InF. Moreover, it was difficult to solve together tangleness and fatness, in order to improve all the Quality Indexes, that solving fatness could instead increase the QDI of InF.

AS refactoring do not affect the SQALE Index that remains constant at the highest value *A*, before and after removing the AS in the systems. Stability Index of SA4J remains constant too, except for one system. Hence, these two indexes are not related and not affected by the refactoring of AS and Tangles. The two indexes could be excluded in order to evaluate some kind of architectural debt.

RQ2 *Can be identified architectural problems that have a higher impact on the Quality Indexes?*

The removal of Fat items led to a higher impact on the QDI of InF. This was observed immediately when the removal Fat classes and methods was attempted. The extraction of classes was attempted to solve a Fat class, but the extracted classes could be Fat classes too and affected by one or more smells. Ignoring this aspect could led to the introduction of new code smells with a negative impact on the QDI.

Hence, the importance to find a compromise on removing AS, Tangle and Fat issues, by taking into account the risk of introducing code smells emerged in this study. It is very important to make the right refactoring choices, and understand if it is better to remove fatness and improve SoC Index of S101 or viceversa improve the QDI of InF and not the fatness.

In conclusion, S101 was very useful to remove AS, as Tangles, and inFusion if it needed to remove not only AS, but also code smells, since the tool is able to detect 21 code smells. Since AS can be considered more critical than code smells, one can focus first the attention on their removal. The AS refactoring could led obviously to the introduction of other problems, but different code smells could be also removed. In this direction, some works have analyzed the possible correlations existing among code smells and AS [121]. Whether these correlations are known and exploited, they could led to some benefits in identifying the best refactoring strategies.

I will address this topic on code smells and architectural smells correlations in Chapter 6.2.

4

ARCHITECTURAL SMELL DETECTION THROUGH ARCAN

Arcan is a tool developed for architectural smells (AS) detection for Java project. The detection is focused on architectural smells based on dependency issues, since components highly coupled and with a high number of dependencies are hard to maintain and hence can be considered more critical. This kind of AS can violate the modularization principle, that advocates the creation of cohesive and loosely coupled abstractions through techniques such as localization and decomposition. The detection of dependency-based AS could establish to what extent different dependency measures or indexes (obtained through the AS detection) could generate different maintenance costs. Arcan detects 6 different architectural smells using both architectural and historical data of projects. The historical data are useful to evaluate the evolution of architectural smells during their life-time. This Chapter is composed by three main sections: the explanation of the architecture of the tool is described in Section 4.1, the architectural smells detected by Arcan are described in Section 4.2 and the conclusions of this Chapter are described in Section 4.3.

4.1 ARCHITECTURE OF ARCAN

The Arcan architecture core consists of four components (see Figure 4.1), structured in a layered architecture style: a user interface built with Java FX (*Presentation Layer*), a main processing unit with all the logic components (*Domain Layer*) and a graph database accessed through a graph computing framework (*Persistence Layer*). The tool is written in Java 8. As shown in Figure 4.1, Apache Tinkerpop[1], the framework which interfaces the *Graph Database*, is used for two reasons: to easily build and access the dependency graph which represents the analyzed project and to allow the exploitation of different graph database backends. This means that all the graph elements, e.g., nodes, are Tinkerpop elements. Every read or write operation that regards the database is filtered by the framework. Hence, the queries are written in Gremlin-Java, the variant of the Gremlin[2] query language that allows to write graph traversals within the native Java environment. Tinkerpop deals with the translation of the dependency graph into specific backend's graphs and hides the underneath database. In the current version of Arcan, the graph generated through Tinkerpop can be stored *1)* in-memory or *2)* using a Neo4j[3] graph database; other backends can be added in the future. Neo4j was selected because it offers an intuitive graphic interface

1 http://tinkerpop.apache.org/
2 http://tinkerpop.apache.org/gremlin.html
3 http://neo4j.com/

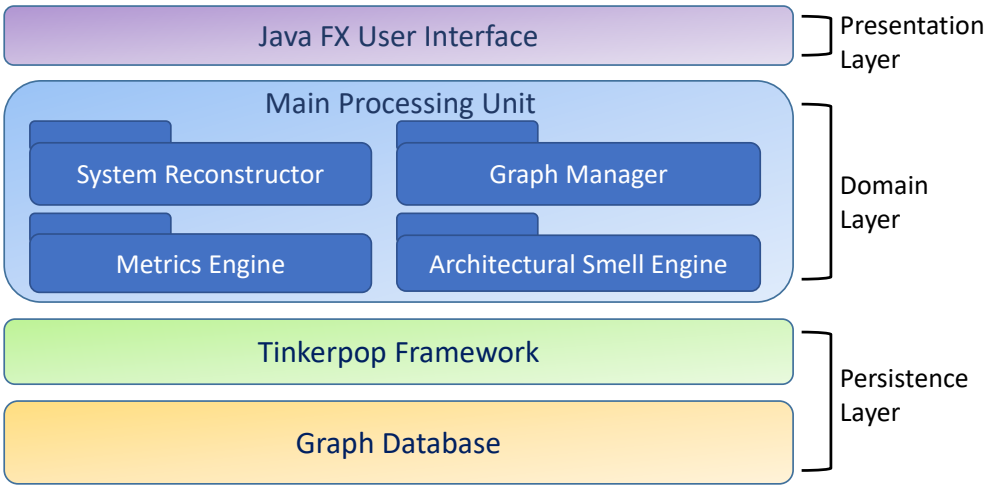


Figure 4.1: Arcan Architecture

which allows the exploration (using Cypher[4] queries) and visualization of the dependency graph built by Arcan. The graph can be browsed to understand the structure of the system and different algorithms can be applied on it to extrapolate more detailed information. After the execution of these algorithms, new nodes and edges are added to the graph as "smell" nodes, which indicate the presence of an architectural smell in the system.

In the following paragraph it is described how the components of Arcan belonging to the *Domain Layer* (Figure 4.1) interact for the detection of the architectural smells without exploiting the history data.

1) The *System Reconstructor* reads the compiled Java files, which can be submitted as a folder of `.class` files or a folder of `.jar` files. Arcan only retrieves classes and packages which are included in the input, without extending the analysis to external components. Hence, to make Arcan analyze a complete project, it is necessary to have every component as input. The information contained in the compiled file is extracted thanks to the Apache Byte Code Engineering Library (BCEL[5]). This library offers a class named `JavaClass` to represent the data structures, constant pool, fields, methods and commands contained in a typical Java `.class` file.

2) The *Graph Manager* is the component dedicated to build the dependency graph. From the `JavaClass` object extracted by the *System Reconstructor*, it is possible to know the system classes, packages and references which link to them. These elements are all included in the dependency graph through Tinkerpop. This component also manages the initialization of the database and writes the dependency graph in it.

4 https://neo4j.com/developer/cypher/
5 http://commons.apache.org/proper/commons-bcel, Apache BCEL 6.0

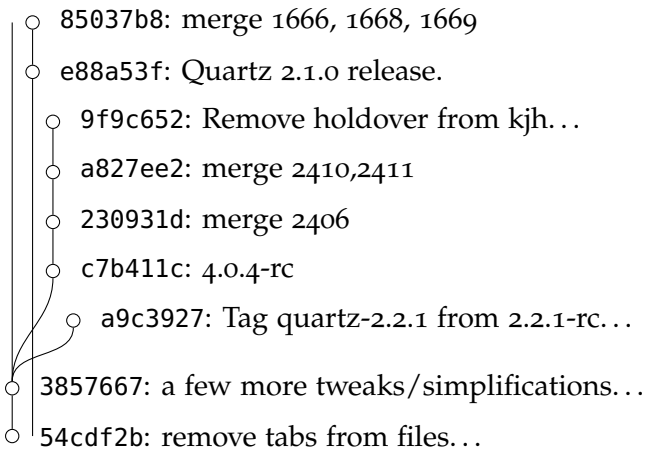


Figure 4.2: Example of Git history log

3) The *Metrics Engine* computes the R. Martin's dependency metrics [109], used in the detection of the architectural smells. Moreover, this engine is entrusted with computing typical cohesion and coupling metrics at the class level, such as Fan In, Fan Out, CBO and LCOM [38]. To compute these metrics, this component accesses the dependency graph and the results are stored as attributes in the nodes representing the classes or packages that the metrics refer to.

4) The *Architectural Smell Engine* contains the logic for both architectural smell detection and filtering of false positive instances. Every detection algorithm extracts a subgraph from the whole dependency graph and works on it depending on the elements which can be affected by the anomaly: classes or packages. When a smell is detected, a new node of type "smell" (called "supernode") is created and linked to the nodes involved in the detection. This makes easier to filter the results in a second step, when necessary.

The detection workflow of architectural smells using the historical data is slightly different from the previous one, since it is based on Java file instead to `JavaClass` and it takes in input the Git[6] log, as showed in Figure 4.2. Git was chosen as starting version control system since old projects are moving to Git [136] and new projects are starting using it[7].

1) The *System Reconstructor* reads the Git log and extracts data about modifications made to Java files by commit. Packages are extracted from the Java file package definition in the head of the file.

2) The *Graph Manager* is the component dedicated to build the dependency graph. From the `Commit` and the `JavaFile` extracted by the *System Reconstructor*, it is possible to know the system packages and references linked to them. Commits sequences will be linked together

6 https://git-scm.com/, `git` - a free and open source distributed version control system
7 Github own more then 68 millions of projects using Git as version control system

as in a time series of events where the oldest event is the starting point of the time series, e.g., the first commit `54cdf2b` will be linked to the second commit `3857667` and so on, as shown in Figure 4.2.

3) The *Metrics Engine* counts the number of times where a change among files in a commit was detected. To detect the smell, it counts the number of changes done among all Java files, and selects the set of files which are most frequently changed together and do not belong to the same package as instances of the smell.

4) The *Architectural Smell Engine* contains the logic for *historical* architectural smell detection and filtering of false positive instances by evaluating the filters defined for the AS. Every detection algorithm extracts a subgraph from the whole dependency graph and works on it depending on the elements which can be affected by the anomaly: classes or packages. When a smell is detected, a new node of type "smell" (called "supernode") is created and linked to the nodes involved in the detection. This makes easier to filter the results in a second step, when necessary.

4.2 ARCHITECTURAL SMELLS

As previously introduced, Arcan detects 6 architectural smells that may be causing instability issues, where instability refers to the predisposition of objects to change [109]. The instability is computed as the ratio between efferent dependencies over the total number of dependencies, where the efferent dependencies are the number of classes inside the package that depend upon classes outside the package. In the following, a subsystem (component) is a set of packages and classes which identifies an independent unit of the system responsible for a certain functionality. Following in this section, the complete explanation of each architectural smell is given, specifying: the *definition* of the architectural smell, the *granularity* of the architectural smell (e.g. if affect classes, package), the *violated design principle*, the description of the algorithm used for the *detection* of the architectural smell and eventually a *filter* defined in order to enhance detection of the architectural smells and reduce the number of false positives in the results.

4.2.1 *Cyclic Dependency (CD)*

Definition: A *Cyclic Dependency* refers to a subsystem (component) that is involved in a chain of relations that breaks the desirable acyclic nature of a subsystem's dependency structure. The subsystems involved in a dependency cycle can be hardly released, maintained or reused in isolation [108].

Granularity: Detected on classes and packages.

Violated Design Principle: Acyclic Dependencies Principle [109].

Detection: To accomplish the detection, the tool relies on a Depth First Search (DFS) algorithm [142] through the following steps: *1)* Extracting the subgraph relative to the requested granularity level (class or package); *2)* Launching the DFS algorithm on the subgraph.

Shapes: For what concerns this smell and enrich its detection, it is useful to refine the detected cycles according to their shape [2] (shown in Figure 4.3). Rules are established 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.

Filters: a possible false positive occur when the instance of the smell is related to the design pattern: Factory Method, as outlined in [19].

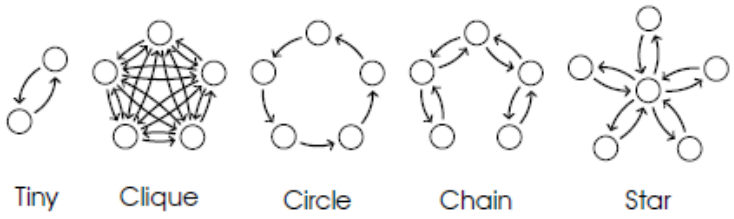


Figure 4.3: Cycles shapes [2]

4.2.2 *Unstable Dependency (UD)*

Definition: *Unstable Dependency* describes a subsystem (component) that depends on other subsystems that are less stable than itself, according to the Instability metric value [109]. This may cause a ripple effect of changes in the system [108].

Granularity: Detected on packages.

Violated Design Principle: Stable Dependencies Principle [109]

Detection: The steps to identify this smell are three: *1)* obtaining from the graph all the dependencies between packages; *2)* computing the Instability metric for every package of the system; *3)* for every package, checking if it is afferent to a less stable package.

Filters: A filter is defined 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. Considering "Bad Dependency" a dependency that points to a less stable package, a formula is applied to establish the "Degree of Unstable Dependency" (DoUD)[19]:

$$DoUD = \frac{BadDependencies}{TotalDependencies} \tag{4.1}$$

A package with a small number of bad dependencies may not be a smell, and this formula helps to filter misleading results. The Degree of Unstable Dependency under which a package is no more a smell can be considered as a *threshold* of the filter. The filter threshold establishing the minimum level needed to consider a package affected by the smell (DoUD) was set at 30%, since this value highlights the largest share of correct UD instances, according to a manual validation I performed while exploring different thresholds over several projects. The different values of bad dependencies can be used also to identify the UD more critical respect to other. This kind of "Severity Evaluation" is exploited in Chapter 7.

4.2.3 *Hub-Like Dependency (HL)*

Definition: *Hub-Like Dependency* arises when an abstraction has (outgoing and ingoing) dependencies with a large number of other abstractions [149].

Granularity: Detected on classes and packages.

Violated Design Principle: "High cohesion and low coupling" is the basis for effective modularization. Meyer's "few interfaces" rule for modularity says that "every module should communicate with as few others as possible." [113].

Detection: The steps to identify this smell are the following: *1)* finding all nodes of type class; *2)* for all of them, calculating the ingoing and outgoing dependencies; *3)* calculating the median of the number of ingoing and outgoing dependencies of all the classes of the system; *4)* checking if the number of ingoing and outgoing dependencies of a class is respectively greater than the ingoing median and outgoing one (i.e., the median helps to normalize the values and exclude small HL); *5)* checking if the difference between ingoing and outgoing dependencies is less than a quarter of the total number of dependencies of the class; if so, the class could be a HL.

The last step defines that the difference between ingoing and outgoing dependencies must be small to consider dependencies as "balanced", as shown in Figure 4.4. After exploring different settings, in the algorithm this quantity was set at 1/4 of the sum of ingoing and outgoing dependencies.

The same detection process is applied at the package granularity level.

Filters: HL are highly used classes and packages of the project. Assuming it is not know if those classes are used from other projects (since Arcan analyzes one system at a time), whether the class uses external classes of the project, 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. Moreover, a false positive HL is detected in classes where it is implemented a Singleton design pattern or where nested (hidden) classes occur, as outlined in [19].

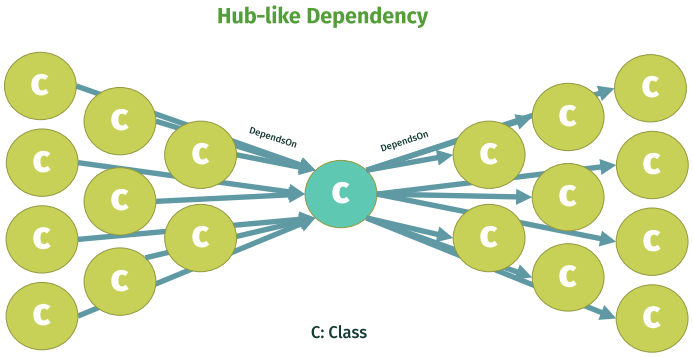


Figure 4.4: Hub-Like Dependency smell example (on classes)

4.2.4 *Specification-Implementation Violation (SIV)*

Definition: *Specification-Implementation Violation* is able to point out architectural violations, since it captures whether the intended architecture is different from its actual implementation. As shown in Figure 4.5, Classes and Packages are grouped by components (C) and the components are linked by constraints edges according to the architecture specification. The detection consists in checking the definition of the intended architecture, represented with rules and constraints in the dependency graph.

Granularity: Component level.

Violated Design Principle: this smell violate *1)* the original, intended architecture of a system or *2)* general software modularity principles (Perry and Wolf [130]).

Detection: this architectural smell needs that components and constraints given and manually defined to Arcan. SIV is detected through the following steps : *1)* definition and creation of components nodes; *2)* link of the classes and packages belonging to a component; *3)* link of the components using the "constraint" edge *4)* check if constraints among two or more components are violated. Figure 4.5 shows an example of SIV smell where the class `C2` violates the constraints among the components `C01` and `C02` since it depends on the class `C1`.

Filters: filters are not defined for this architectural smell.

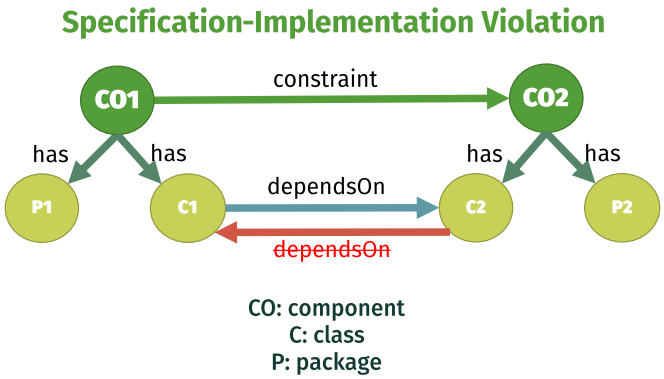


Figure 4.5: Specific-Implementation Violation example

4.2.5 *Multiple Architectural Smell (MAS)*

Definition: *Multiple Architectural Smell* identifies a subsystem (component) that is affected by more than one architectural smell and provides the number of the architectural smells involved.

Granularity: Detected on classes and packages.

Detection: the architectural smells detection of the other AS must be performed before MAS smell detection, for obvious reasons. The detection of this smell is performed through the following steps: *1)* Arcan gets all the AS of the project; *2)* for all AS retrieves all the classes or packages involved in at least two AS.

4.2.6 *Implicit Cross Package Dependency (ICPD)*

Definition: *Implicit Cross Package Dependency* is a history based architectural smell defined to compute the degree of co-changes occurring among files belonging to different packages detected by analyzing the change history. This smell has been introduced and studied also by Kouroshfar et al. [76] and Mo et al. [114], but the detection of the smell is done in a different way: in this case the detection is based on the graph dependency model, in their case on the Design Structure Matrix (DSM). The *Implicit Cross Package Dependency* smell captures hidden dependencies among files belonging to different packages. Hidden dependencies are co-change relations that can be found only in the history of the project. Files changed frequently together with hidden dependencies lead to a lack of modularity.

Granularity: Detected on files.

Violated Design Principle: Stability Principle [23], Baldwin et al. [23] said that the design rules have to be stable, that is, neither error-prone nor change-prone. Moreover, If two packages are truly independent, then they should only depend on design rules, but not on each other. More

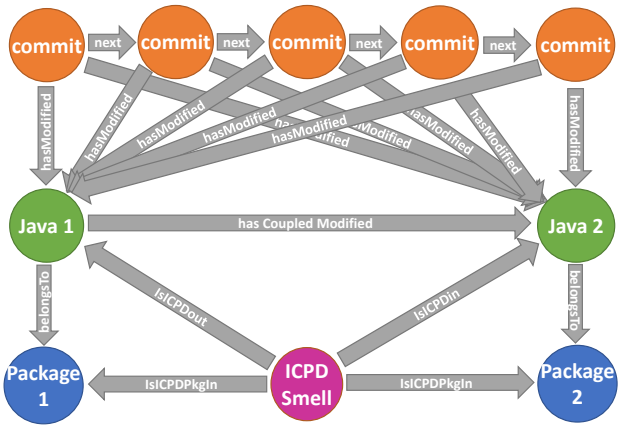


Figure 4.6: Implicit Cross Package Dependency example

importantly, independent packages should be able to be changed, or even replaced, without influencing each other, as long as the design rules remain unchanged.

Detection: The detection is done through the analysis of the graph representation of the history of the project development. The history of the development is stored in the version control system. As outlined before, projects using Git as version control system are the only analyzable with Arcan. To detect the smell, the number of changes (by commit) done among all Java files are used to select the set of files which are most frequently changed together and do not belong to the same package, as instances of the *ICPD* smell. The detection is performed through the following steps: *1)* all the commits are ordered in temporal order, from the oldest to the most recent; *2)* a commit node is created for all commits retrieved. Java files modified in that commit are retrieved. If Java files are not modified in the commit, the algorithm jumps to the next commit; *3)* creates nodes of all Java files modified by the commit. Otherwise, if they are already present, the Java files will be linked to the new commit. *4)* the Java files mostly modified together but not belonging to the same package are an ICPD smell.

Figure 4.6 shows an example of Implicit Cross Package Dependency where two Java files are changed together in five commits. `Java 1` belongs to the package `Package 1`, while `Java 2` belongs to the package `Package 2`. I defined the following four node types:

- `commit`: represents a Git commit and it is related to all the modified Java files using the edge `hasModified`.
- `java`: Java file representation node. It can be related to another Java file using an edge of type `hasCoupledModified`, if these two files are modified together (co-changed) in at least a commit. A Java file is linked to a package node with an edge of type `belongsTo`.
- `package`: node indicating a package.

- `smell`: represents the architectural smell Implicit Cross Package Dependency. It is related to two Java files and two packages, using four edges. `isICPDin` and `isICPDout` refer to Java files and `isICPDPkgin` and `isICPDPkgout` refer to packages.

Filters: Two measures are defined to avoid some false positives cases, called *Strength* and *Support*, defined as follows:

- *Strength* is the ratio of commits where two files are co-changed together over the total changes of the single file. To consider two files affected by ICPD smell, the Strength value must be higher than a defined threshold.
- *Support* is the number of co-changes of two files. To consider two files related to ICPD smell, Support must be higher than a defined threshold.

I performed a study (described in Section 5.1.4) to determine how the thresholds values of Strength and Support affect the detection. The instances of Implicit Cross Package Dependencies with different combinations of Strength and Support values have been detected. Strength has been varied between 0.5 and 1 with steps of 0.1, and Support has been between 2 and 10 with steps of 1. Strength and Support have been selected respectively as 5 and 0.6 according on the evaluation performed in the study.

4.3 CONCLUSIONS

In this Chapter I introduce Arcan tool, its architecture and the 6 detectable architectural smells. Arcan aids the detection and the removal of architectural smells, hence it is useful to support software developers and designers during the development, maintenance and evolution of Java application.

The tool relies on graphs to represent the extracted information, the dependency graph. The application of graph databases makes the graph reusable for further analyses, including the experimentation of new algorithms, and the implementation of new detectors. Once a Java project has been analyzed by Arcan, a new graph database is created containing the structural dependencies of the system. Thanks to graph computing and connected big data processing technology [158], it is then possible to run AS detection algorithms on this graph to extract information about the analyzed project. The applied graph computing technology allows the exploitation of different graph database backends, while relying with a flexible API. This represents a recent trend in the literature [43].

Many tools have been developed for code smells detection but only few tools are currently available for architectural smells detection, as described it Section 2.2. Arcan detects two new architectural smells: Multiple Architectural Smell and Specification-Implementation Violation. The detection of the other architectural smells is enhanced w.r.t. previous approaches. In particular, Hub-Like dependency is also detected at package level, Cyclic

dependency smell is detected in according to different the cyclic shapes [53, 93] and Unstable dependency exploits a validated metric to avoid false positives. Moreover, Arcan allows to perform history-based analysis to detect ICPD smell having as input the Git log of the project. The detection of ICPD relies on the dependency graph, as differently made in other approach [114]. Finally, the dependency graph provided by Arcan is useful to identify the possible refactoring opportunities of an architectural smell and the metrics used in the detection techniques of the architectural smells can be exploited to identify the most critical ones (as explained in Chapter 7).

The detection of the architectural smells has been validated, as explained in Chapter 5, and filters has been proposed to reduce the false positive instances of AS.

On this study regarding the Arcan development I co-supervised the bachelor thesis of Ilaria Pigazzini and two papers have been published [17, 19].

5

EVALUATION AND VALIDATION OF ARCAN RESULTS

This Chapter describes the evaluations and the validations of the Arcan tool presented in Chapter 4 and it outlines: a first evaluation of Arcan results on different open source projects in Section 5.1, the validation on two industrial projects 5.2 and a mixed-method study of architectural smell validation in Section 5.3.

5.1 INITIAL EVALUATION OF ARCAN RESULTS

This study outlines the results retrieved from the analysis on seven open source projects, listed in Table 5.1 for three AS: Cyclic Dependency, Hub-Like and Unstable Dependency. The *Implicit Cross Package Dependency* smell has been validated in a different way since it is the only AS whose detection is based on the history of the project' development.

After every analysis Arcan writes the results in six *csv* files. The execution of both Unstable Dependency and Hub-Like smell detectors generates one file, while for Cyclic Dependency two files are generated. The last files in the results pool are the ones containing the computed Martin metrics [109] for packages and classes. These data are obtained by simply applying the algorithms derived from the definitions of the three architectural smells described in the previous chapter.

Moreover, we have analyzed the differences in the detection results obtained by Arcan with respect to other two tools, inFusion [67] and Hotspot [114]: inFusion [67] for the detection of Unstable Dependency and Hotspot for the detection of Cyclic Dependency. The results have been manually checked by three evaluators (PostDoc, PhD and MsC students).

Hence, the aim of the comparison of Arcan results with those of the other two tools was done only to better check and improve the results of Arcan, that led to the introduction of some Filters. This study do not aim to validate

Table 5.1: Analyzed Projects

Project	Version	Lines of Code (LOC)	Num of Packages (NOP)	Num of Classes (NOC)
Derby	10.9.1.0	651118	217	3010
Jedit	4.3.2	109515	38	1017
Junit	4.10	6580	28	171
Maven	3.0.5	65685	143	837
Quartz	1.8.6	24522	39	455
Spring	3.0.5	329358	598	4615
Struts	2.2.1	143196	258	2231

Table 5.2: Unstable Dependency Results

	Affected Packages	
Project	**Arcan**	**inFusion**
Derby	35	5
Jedit	14	2
Junit	11	0
Maven	69	1
Quartz	8	1
Spring	107	5
Struts	43	7

the results of the tool in terms of accuracy, also because there is no reference data or established evaluation criteria in the literature to be applied.

5.1.1 *Unstable Dependency Smell Results*

The output file of this smell consists in a table showing the packages affected by the smell, the packages which cause the smell and their respective Instability. Table 5.2 displays the results of Unstable Dependency's detection on the seven projects, starting from the definition of the smell given before.

Table 5.3 instead shows the results retrieved from the analysis of Apache Quartz, which was manually analyzed and inspired the proposal of a Filter. Quartz is chosen as an example of medium size project. The first column contains the packages which Arcan considers affected by the Unstable Dependency smell; the second and the third columns compare the detector's results with the InFusion tool's ones; the fourth column shows the results of the application of the Filter described in Section 4.2.2, using a threshold ("bad" dependency ratio) of 30%; the last column shows the new results after the filtering process. Arcan results agree with InFusion's ones on a single package, *org.quartz.utils*, the only one with every 100% of bad dependency. In this example, the Filter threshold was set at 30%, since it highlights the largest share of correct UD instances, according to the manual validation performed. Filters are explored with different thresholds. Figure 5.1 shows the percentage of packages detected as UD in all the projects, when varying the threshold from 10% to 100%. As we can see in Figure 5.1, the number of detected packages is almost stable with thresholds $\geq 50\%$.

In conclusion, from Table 5.3 we can observe that Arcan detects a larger number of UD with respect to inFusion, since inFusion detected only one Unstable Dependency on `org.quartz.utils.counter` package.

Table 5.3: Quartz Unstable Dependency Results

Package	Arcan	inFusion	Bad dep.%	Filtered
`org.quartz.core`	yes	no	<30%	no
`org.quartz.utils.counter.sampled`	yes	no	35%	yes
`org.quartz.ee.jta`	yes	no	32%	yes
`org.quartz.impl`	yes	no	<30%	no
`org.quartz.utils`	yes	yes	100%	yes
`org.quartz.impl.jdbcjobstore.oracle`	yes	no	<30%	no
`org.quartz.utils.counter`	yes	no	83%	yes
`org.quartz`	yes	no	60%	yes

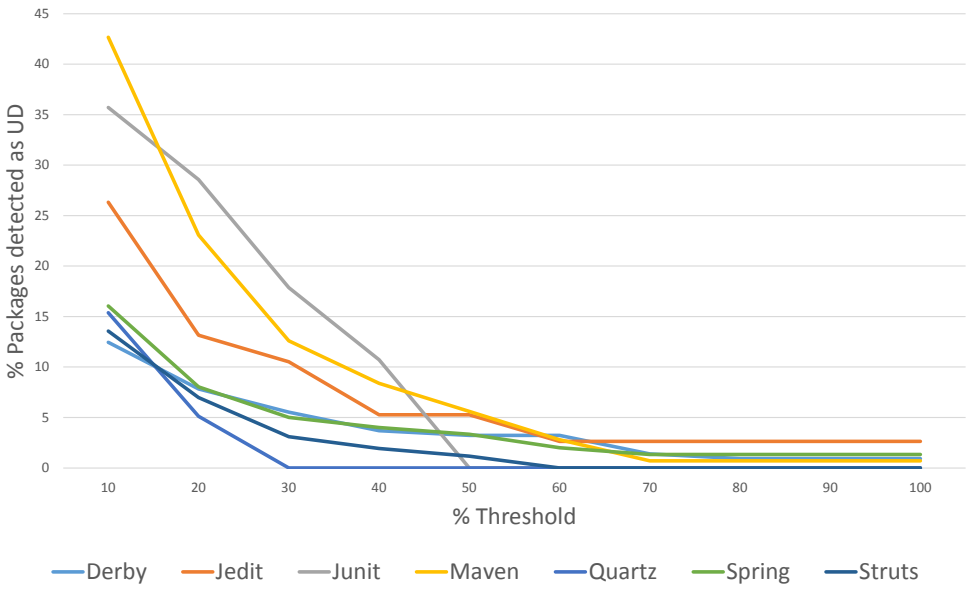


Figure 5.1: Filtered Unstable Dependency Results

5.1.2 *Hub-Like Results*

The detection of this smell produces a file containing the list of the detected HL classes. The results retrieved from the analyzed projects are displayed in Table 5.4. For this smell, it is not provided a comparison with other tools, since it is not known of any tool detecting it.

No Filters are needed for this smell. In fact the detection algorithm not only considers classes with a large amount of dependencies in general, but also classes with a balanced number of ingoing and outgoing ones; these are

Table 5.4: Hub-Like Results

Derby	Jedit	Junit	Maven	Quartz	Spring	Struts
1	7	1	3	0	2	1

Figure 5.2: Example of Junit Hub-Like class

Table 5.5: Cyclic Dependency Results

Project	Couples	Couples not-duplicated	Hotspot
Derby	5520	736	92
Jedit	552	131	42
Junit	110	70	22
Maven	4036	749	32
Spring	0	0	4
Struts	2160	558	54

the two characteristics which make them HLs. See Figure 5.2 for a graphical example of Hub-Like smell, obtained from the graph database.

While performing our study on the Hub-Like architectural smell, it was evident that there are two possible interpretations of this smell, depending on whether all classes with a reference from/to the HL are internal to the project or not. In the first case, the HL is defined until now, as an architectural issue. In the second case, for instance if the HL has dependencies equally divided in internal ingoing and external outgoing ones, it could be seen as a feature of the architecture instead of an issue. This happens since the HL class could have been chosen as a controlled exit point to logically divide the internal project from the external libraries. It is not excluded a third case where an HL with a mixture of internal and external ingoing/outgoing dependencies could be a problem for of its lack of architectural logic.

Table 5.6: Cyclic Dependency Results

Packages	package1	package2	package3
cycle1	1	1	0
cycle2	1	0	1

(a) CD results list

Packages	package1	package2
package1	5	3
package2	3	6

(b) CD results matrix of package

5.1.3 *Cyclic Dependency Results*

To better manage the results of Cyclic Dependency (CD) detection, two different files offer two points of view of this smell. The first one contains a table with cycles as rows and class/packages as columns (see Table 5.7b): in this way the focus is on the cycles and the elements (packages/classes) which make it up. The second file consists in a matrix with the same elements on rows and columns and with cells filled with counters (see Table 5.7a). These counters refer to the number of times a couple of elements were found in the same cycle. For instance, if two packages are present together in three different cycles, the cell corresponding to the respective row and column will contain the number 3, as shown in Table 5.7b among the `package1` and `package2`.

The tool detects cycles relying on the DFS algorithm (see Section 4.2.1), retrieving cycles of different sizes, where often bigger ones contain smaller ones. A proposal of results refinement could be to consider only the bigger ones and discard the others, to refine the results and obtain clearer information. Detecting cycles having size ≥ 2 implies that more (and less obvious) cycles will be detected than considering size $= 2$. Table 5.5 contains the number of couples of packages found in cycles retrieved from the second output (the one in matrix form) of the CD algorithm: column "couples" contains all couples that were found together in the same cycle, and considers their repetition caused by their presence in more than one cycle; column "couples not-duplicated" shows the number of couples in cycles too, but counting each one only once. The last column reports Hotspot Detector [72, 114] results, which considers only cycles having size $= 2$, i.e., couples. The differences in the tools' results are justified by their choice of the size of the cycle to detect.

In conclusion, from Table 5.5 we can observe that Arcan detects a larger number of CD (i.e. Couples not-duplicated) with respect to HotSpot, since Hotspot detects only cycles having size $= 2$.

5.1.4 *Implicit Cross Package Dependency Results*

The previous validations were performed on architectural smells not related to the development history, i.e., the AS consider a single version of the

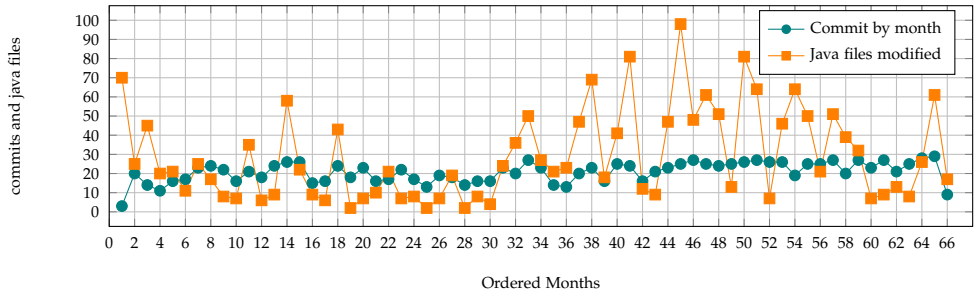


Figure 5.3: Commit of Tomcat by month and number of modified Java files

Figure 5.4: Max ICPD of Tomcat by month

project. Arcan detects only an AS considering the history of the project's development, the *Implicit Cross Package Dependency* (ICPD) smell.

The ICPD validation is done through the evaluation of the evolution of the smell in two open source projects, in order to study the metrics used for the computation of the smell: Strength and Support defined in the previous chapter. As an example, it is shown in Figure 5.4-5.6 the trends of the *Implicit Cross Package Dependency* smell in two systems: JGit and Tomcat. It is evident that the architectural smell is highly present in both the systems, but with different trends.

Figure 5.3-5.4 shows the number of commits and the evolution of the ICPD instances in Tomcat on a monthly basis. The number of ICPD instances is clearly higher in the last third of the development software history, with large differences between subsequent months. However, it was evident through the analysis done that such a high number of files changed together was due to actions not related to the software maintenance or development. For

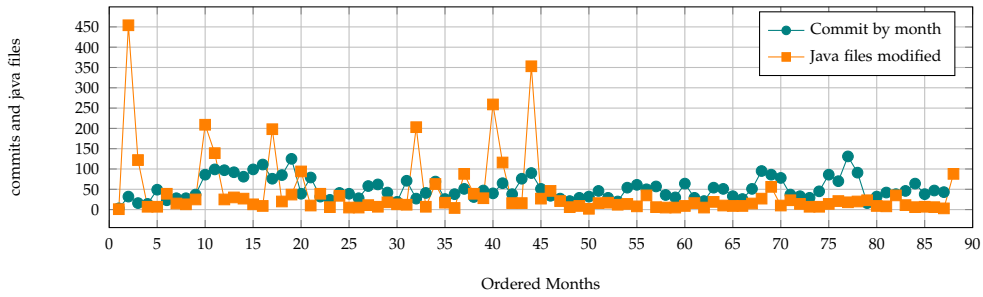


Figure 5.5: Commit of JGit by month and number of modified Java files

Figure 5.6: Max ICPD of JGit by month

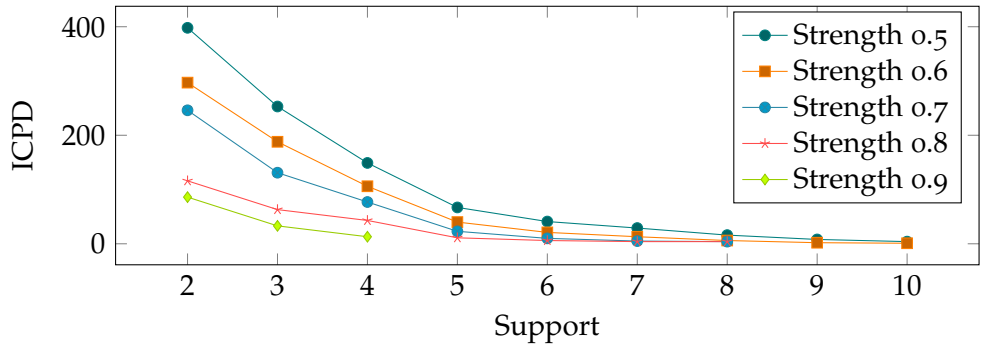


Figure 5.7: ICPD trend changes for Support and Strength in Tomcat

example, one commit generating the smell is related to the change of the software distribution licence: developers changed all source code files in batch, by injecting or replacing a comment in the header of the file. This large file co-change increased the *Support* and *Strength* measures over their thresholds, generating the smell.

Figure 5.5-5.6 shows the same data for JGit. Despite the presence of spikes in the graphs, there is an increasing trend until the 70th month and a decreasing trend starts after this point. From these data, it is observable that ICPD is not influenced by commit activity, since many time-spans with high commit activity do not result in an increase of ICPD instances. Moreover, while the main trend is to increase the number of ICPD instances, they can also be removed over time, with spikes in the graphs witnessing peculiar situation, e.g., very large commits due to different non-development related reasons. An ICPD instance is removed over time by changing its files separately in different commits. In this way, the *Strength* characterizing the instance returns under the threshold and the smell is removed. This is the symptom of what happens if a developer fixes the implicit dependency: the detector will not notice the fix until the involved files start evolving independently.

Figures 5.7-5.8 show the number of Implicit Cross Package Dependency with different configurations of *Support* and *Strength* on the last version of both systems. There are not architectural smells of this type with *Strength* equal to 1 in both systems. It is evident that *Support* fixed at 3 reduces significantly the number of architectural smells detected in both systems. Architectural smells in JGit are not present with *Strength* equal/greater than 0.8 and *Support* greater than 2. Through manual inspection of the results con-

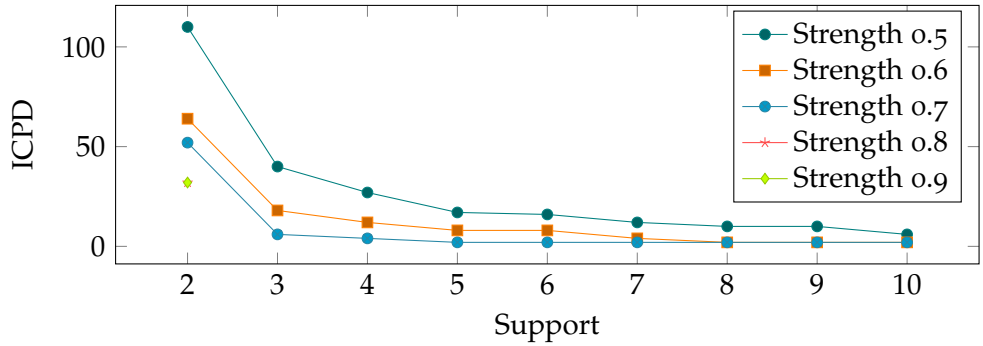


Figure 5.8: ICPD trend changes for Support and Strength in JGit

sidering *Strength* greater than 0.6 and *Support* greater than 2, it was possible to filter out the false positives related to the first commit, a short sequence of similar commit and the files changed together but never modified again.

5.2 ARCHITECTURAL SMELLS VALIDATION: AN INDUSTRIAL CASE STUDY

Section 5.1 shows the results of Arcan in different open source projects. This section describes an evaluation of Arcan results performed by *external tool developers* to check if the architectural smells detected by Arcan are really perceived as problems and to get from them an overall usefulness evaluation of the tool.

The following two projects have been analyzed:

1) DICER[1]: a continuous architecting tool for data-intensive applications (DIAs) that allows to quickly put together a model of a data-intensive application using known DIA middleware such as Apache Spark, Apache Hadoop MapReduce and Apache Storm.

2) Tower4Clouds[2]: a flexible, self-adaptable and auto-configurable monitoring infrastructure engineered for multi-cloud applications. Tower includes multiple data-collectors that allow monitoring, collecting and sifting from multiple data sources by means of a rule-based approach.

The evaluation was carried out by direct observation of Arcan results. In particular, for each project under study, 3 professional software designers experienced with the projects discussed (separately) the Arcan evaluation-sheets line-by-line, quickly checking the code and/or via available documentation and deliverables to confirm/refute Arcan findings. These practitioners reported on: *(a)* whether Arcan actually uncovered known or unknown architecture issues; *(b)* whether the issues were actually issues; *(c)* whether refactoring was needed or planned following Arcan results. Reports were captured using in-line comments directly on Arcan result plots – this data is freely available online[3] to encourage verifiability.

Size metrics for the projects are shown in Table 5.7. The total number of AS in the projects and the evaluation of Arcan detection performances is

1 https://github.com/dice-project/DICER
2 http://deib-polimi.github.io/tower4clouds/docs/overview.html
3 http://tinyurl.com/zpquemg

Table 5.7: Analyzed Projects

Projects	DICER	Tower4Clouds
Version	0.1.0	0.3.1
Packages(NOP)	549	373
Classes(NOC)	13204	8820
Analyzed Component	it.polimi.dice.dicer	it.polimi
Packages(NOP)	9	7
Classes(NOC)	36	111

Table 5.8: 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 5.9: 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

reported Table 5.8. Standard Information Retrieval performance metrics are reported, i.e., confusion matrix elements and derivatives, like precision and recall.

The tool obtained a precision of 100%, since Arcan found only correct instances of architectural smells but developers reported 5 more architectural smells which are False Negatives. False Negatives were related to external components out of the scope of the analysis of the tool.

As can be seen from Table 5.7, the developers focused their attention on a component of the projects, and not on the AS found in the entire projects: this is caused by the high number of the detected AS. In particular, for the CD smell Arcan found 636 occurrences in DICER and 439 in Tower4Clouds (see Table 5.9). As a consequence, using followups and feedback from the evaluation we decided to define a Severity Index for the CD smell; the purpose of the index is to assist in the identification of the most critical smells to be analyzed and then removed first. The *Severity Index* counts the number of vertices involved in a cycle and the weight (number of occurrences) of every edge which forms the cycle, then orders the instances of Cyclic Dependency descending by the number of vertices in the cycle and the maximum number of times the cycle occurs. This index is better defined and used in the computation of the Architectural Debt Index described in Chapter 7.

Important hints were gotten from this evaluation on how to improve Arcan results to remove possible false positive instances: (a) the detected false positives for Hub Like Dependency smell reflect abstract classes, interfaces and classes which implement the Singleton design pattern; (b) the Cyclic Dependency smell false positives reflect classes which implement Factory Method design pattern and nested (hidden) classes.

5.3 ARCHITECTURAL SMELLS VALIDATION: A MIXED-METHOD STUDY

Arcan was initially experimented on several open source projects [17] (see Section 5.1), then a more careful qualitative evaluation of Arcan results was performed through external tool developers [19] on two industrial projects described in Section 5.2.

The goal of the study described in this section done in collaboration with a colleague of the Politecnico of Milano [135], is to analyze the ability of Arcan to *completely*, and *correctly* detect architectural smells through a quantitative evaluation approach based on understanding to what degree Arcan detects architectural smells previously identified in an independently-coded dataset [22]. The mixed-methods connotation of this part of the study rests on the deliberate choice to use a human-coded dataset to assess the quantitative discoveries operated by Arcan, using an assessment score (i.e., Krippendorff Alpha [79]) typically aimed at content analysis in qualitative research.

The mixed-methods connotation of this part of the study rests on the deliberate choice to use a human-coded dataset to assess the quantitative discoveries operated by Arcan, using an assessment score (i.e., Krippendorff Alpha) typically aimed at content analysis in qualitative research. More details on this feature of the study are outlined below.

The study aims at addressing the following research questions:

RQ1 "Is Arcan exhaustive in architectural smells detection?" - in other words, this research question addresses the completeness of the Arcan tool with respect to its ability to detect *all* instances of architectural smells in a given, independently pre-coded dataset.

RQ2 "Is Arcan correct in its architectural smells detection?" - in other words, this research question addresses the correctness of Arcan detection with respect to its ability to detect the *right* architectural smell in the *right* class (or package), with respect to a given, independently pre-coded dataset.

Answering both research questions positively would essentially amount to stating that the tool detection feature is *correct* and *complete*.

5.3.1 *Study Variables and Data Extraction*

Both research questions 1 and 2 assume that an independently-coded evaluation dataset can be used, featuring projects with architectural issues made explicit by an independent party.

However, for the sake of generalisation, the dataset in question, needs to reflect a series of controllable variables, in order to offer a reliable evaluation. In this case, it is necessary to control the following variables to obtain a viable dataset:

- *Project Size* - The presence of certain architectural smells could be higher in projects with certain size; for this reason was chosen to consider equally both small/medium (200 KSLOC - 500 KSLOC) and large (> 501 KSLOC) projects.

- *Team Size* - The presence of certain organisational dynamics could lead to the emergence of specific architectural smells; we aimed at assessing the validity of Arcan in several team circumstances wherefore technical debt detection makes sense. In this respect, team sizes was controlled across the dataset, to warrant a sufficient coverage of small (<25 participants), medium (20-30 participants) and large (>35 participants) communities.

- *Project Popularity* - Popular projects tend to be subject of continuous releases and continuous refactoring to code; these circumstances may therefore increase the number of architectural smells across the sample such that Arcan correctness in detection may be compromised. In this respect, for what concerns project popularity was controlled that a sufficiently different number of stargazers are present for the projects of the dataset.

- *Project Type* - Very complex products and projects may tend to be more error-prone at the architectural level than simpler projects; these

circumstances may compromise the ability of Arcan to detect architectural smells where code or architectural complexity may compromise project features rather than code size. In this respect it was controlled that the projects in the sample are equally distributed across 5 categories: (1) middleware; (2) software application; (3) development library; (4) application framework; (5) scheduling system.

To address the above data requirements, the literature and open datasets available were investigated from several communities (e.g., MSR, ICSME to name a few). The choice fell on the dataset made available by Asadollah et al. in [22], where the authors analyse over 150+ releases evenly arranged across 5 open-source projects, namely: Apache Hadoop, Apache ZooKeeper, Apache Oozie, Apache Accumulo, and Apache Spark. The projects are summarised in Table 5.10 - the table shows the projects' demographics gathered from https://www.openhub.net, in particular, rows address the controllable variables of the study described previously, while the columns name the specific project.

Table 5.10: Projects demographic data of the dataset

	Apache Projects				
	Accumulo	**Oozie**	**Hadoop**	**Spark**	**ZooKeeper**
Releases	11	6	76	6	20
Size	367.854	194.599	2.434.336	1.207.823	144.322
Popularity	24	1	117	50	22
Team Size	94	29	210	1552	18
Type	(3)	(5)	(1)	(2)	(4)
Activity	High	Moderate	Very High	Very High	Very Low

Project Type: (1) Middleware; (2) Software Application; (3) Development Library; (4) Application Framework; (5) Scheduling System.

Apache Hadoop is the biggest project, the most popular one and its developer team has a very high work activity respect to the other projects of the dataset. Apache Spark has the biggest team size of the dataset with 1552 developers. Apache Ozzie is the less popular of the dataset, but it has a medium team size with a moderate development activity. Apache Accumulo is a medium project size with a large community. Apache Zookeeper is the smallest project of the dataset and it has a small and very low active development team.

For each release of the 5 projects, the authors dig in JIRA[4] issue-tracking information and provide a list of issues and buggy-classes per release, manually mapping every issue to a concurrency bug [22].

In this study, the same dataset was reused in part [5] (all the compile-available versions), as follows.

4 an issue management platform allowing users to manage their issues throughout their entire life cycle

5 openly available online: https://goo.gl/wcdD16

First, every release found on the dataset were cloned. Second, Arcan was runned to detect architectural smells over the release. Finally, the human coding of architecture issues available in Asadollah et al [22] were compared with the architectural smell detection provided by Arcan in order to discover possible overlaps, according to a mapping of every concurrency bug to an architectural smell, as outlined in Table 5.11. The table provides the description of each concurrency bug (first column) mapped to an architectural smell (second column) and the rationale of this mapping (third column).

Table 5.11: Mapping between Concurrency Bugs and Architectural Smells

Bug	*AS*	*Rationale*
Data-race occurs when at least two threads access the same data and at least one of them write the data [166]. It occurs when concurrent threads perform conflicting accesses by trying to update the same memory location or shared variable [1, 61].	CD	Jannesari et Al. show empirically that data races correspond to cyclic dependency scenarios across the architecture [68].
Deadlock is a condition in a system where a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs [26]. More generally, it occurs when two or more threads attempts to access shared resources held by other threads, and none of the threads can give them up [56, 61]. During deadlock, all involved threads are in a waiting state.	UD	The Microsoft Development Handbook identifies recurrent deadlock scenarios as being clearly connected to UD, a scenario which replicates pretty much in an identical manner for Livelocking[6].
Livelock happens when a thread is waiting for a resource that will never become available while the CPU is busy releasing and acquiring the shared resource. It is similar to deadlock except that the state of the process involved in the livelock constantly changes and is frequently executing without making progress [36].	UD	See rationale of Deadlock
Starvation is a condition in which a process indefinitely delayed because other processes are always given preference [148]. Starvation typically occurs when high priority threads are monopolising the CPU resources. During starvation, at least one of the involved threads remains in the ready queue.	any	Architect Michael Barr et Al. explain[7] that Starvation is connected to any condition wherefore the architectural structure does not warrant for proper deadlock and racing freedom.
Suspension-based locking or Blocking suspension occurs when a calling thread waits for an unacceptably long time in a queue to acquire a lock for accessing a shared resource [92].	HL	Empirical practice has shown this mapping recurrently, a most renown case is the Linux Kernel Suspend and "Hibernate-Resume" Blocking[8].
Order violation is defined as the violation of the desired order between at least two memory accesses [69]. It occurs when the expected order of interleavings does not appear [129]. If a program fails to enforce the programmers intended order of execution then an order violation bug could happen [97].	UD	Mair and Herold [103] provide an extensive discussion of this mapping.
Atomicity violation refers to the situation when the execution of two code blocks (sequences of statements protected by lock, transaction) in one thread is concurrently overlapping with the execution of one or more code blocks of other threads such a way that the result is inconsistent with any execution where the blocks of the first thread are executed without being overlapping with any other code block.	HL	Lu et Al. [96] discuss this mapping offering several resolution techniques and design patterns aimed at addressing the issue.

5.3.2 *Data Extracted to Answer the Research Questions*

First, to answer the first research question, namely: "Is Arcan exhaustive in architectural smells detection?" Arcan was evaluated to check if it returned as "smelly", *all* the classes hand-coded as "smelly" by the human colleagues [22].

Second, to answer the second research question, namely: "Is Arcan correct in its architectural smells detection?" the agreement between the hand-coding of architectural smells and the Arcan detection results has been evaluated. To systematically evaluate it, the Krippendorff's Alpha coefficient [79], a widely known inter-rater agreement measure, has been adopted. More in particular, by computing a Krippendorff's Alpha score between an independently coded dataset of issues with Arcan architectural smells, the score average around 0.82 well beyond the standard reference score of $\alpha > 0.800$.

5.3.3 *Empirical Study Results*

RQ1. IS ARCAN EXHAUSTIVE IN ARCHITECTURAL SMELLS DETECTION?
In response to this RQ, Arcan is exhaustive in architectural smells detection, since a total architectural smells match was obtained with architecture issues from the independent, hand-coded dataset. All the architecture issues found in the dataset mapped to an architectural smell, according to Table 5.11, have been detected by Arcan. This mapping has been found independently from the controlled variables defined in Section 5.3.1.

RQ2. IS ARCAN CORRECT IN ITS ARCHITECTURAL SMELLS DETECTION?
In response to this RQ, Arcan is correct in its architectural smells detection, since its prediction overlaps almost completely with the prediction of the human colleagues. As previously stated, Krippendorff's alpha test was used to evaluate the overlap between the two predictions.

Table 5.12 reports Krippendorff's alpha test results performed on all the projects for the mapping between architectural smells and architecture issues from the independently-coded dataset. The median of the tests showed that there is a strong significance with value of 0.842, among architectural smells and architecture issues. The architecture issues less present in the dataset, e.g., Suspension, Livelock, Atomicity Violation Starvation as shown in Figure 5.9, performed the best test results reported at Table 5.12. UD smell scored the best match with Livelock and it performed results grater than 0.88 by all projects. Similarly, MAS smell had a strong significance test values with Starvation bug by all projects. Moreover, HL smell performed good results with values near or above 0.8 for Suspension and Atomicity Violation bugs by all projects. Data Race and Order Violation are performing worst in 4 projects of 5 and Deadlock is the only one with values smaller but

6 https://technet.microsoft.com/en-us/library/ms177433(v=sql.105).aspx
7 https://barrgroup.com/Embedded-Systems/How-To/Top-Ten-Nasty-Firmware-Bugs
8 https://01.org/blogs/rzhang/2015/best-practice-debug-linux-suspend/hibernate-issues

Table 5.12: Krippendorff's Alpha test results for mapping of Bug and AS

AS	Bug	Apache Projects				
		Accumulo	Hadoop	Oozie	Spark	ZooKeeper
CD	Datarace	0.430	0.368	0.419	**1.000**	0.368
HL	Atomicity Violation	**1.000**	0.730	**0.875**	**1.000**	0.730
	Suspension	0.648	**0.842**	**0.875**	0.777	**0.842**
MAS	Starvation	**1.000**	**0.959**	**0.875**	**1.000**	**0.959**
UD	Deadlock	0.783	0.695	0.763	0.777	0.695
	Livelock	**1.000**	**0.882**	**1.000**	**1.000**	**0.882**
	Order Violation	**0.886**	0.662	0.763	0.594	0.662

Legenda: Values grater or equal to 0.8 are highlighted in **bold**

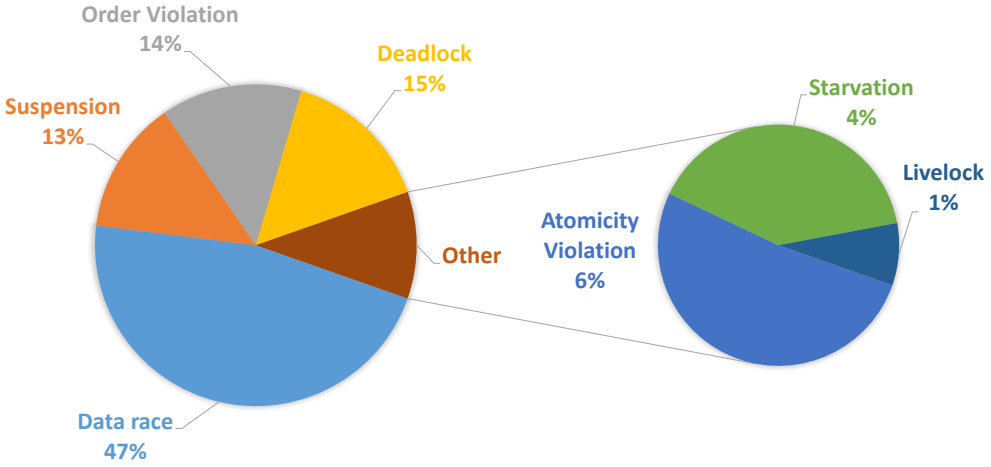


Figure 5.9: Architecture Issues in the Dataset, a pie chart.

near to 0.8. These worst results are obtained on the most frequent Architecture Issues (Data Race and Order Violation), as shown at Figure 5.9.

5.3.4 *Threats to Validity*

Based on the taxonomy in [161], there are four potential validity threat areas, namely: external, construct, internal, and conclusion validity.

"External Validity" concerns the applicability of the results in a more general context. By means of an externally-coded dataset which was prepared previously to the study and in a totally independent fashion, we aimed explicitly to strengthen external validity of this study.

The aim of this dataset usage was to control variables that could warrant a higher external validity, as the variables that were controlled in the several options available for dataset reuse. Although a number of variables remain uncontrolled (e.g., programming language, code structure complexity, PL

type, etc.) due to tool or other limitations, I'm confident to have strengthened the external validity of the current release of Arcan to its highest possible as part of the last evaluation reported in this section.

"Construct Validity" and *"Internal Validity"* concern the generalizability of the constructs under study, as well as the methods used to study and analyze data (e.g. the types of bias involved). The methods adopted to evaluate the solution do not suffer from generalization issues since the evaluation methods essentially re-use established content analysis techniques and inter-rater agreement indexes previously known and widely-used in the literature. Given that the comparison operated in the context of this study relates to the use of independent human observers, we chose to use a simple Krippendorff's Alpha coefficient evaluation. We used a library implemented in *R* called *irr*[9]. More complex inter-rater agreement evaluators exist, e.g., Cohen Kappa and similar. In the future the more advanced techniques will be applied to further assess the validity of the research solution.

"Conclusion Validity" concerns the degree to which the conclusions are reasonable based on the data. The conclusions are based on the manual inspection of the inter-rater reliability scores between a human predictor triangulated independently and in a completely un-related study and the prediction of the research solution. This interpretation is reasonably valid since it is limited to assessing whether the overlap successfully addresses the research questions.

Replication Package

To allow for verifiability of the results, an extensive replication package is provided, containing:

(a) the version of Arcan that was used for the purpose of this study, as well as instructions to setup and run the tool[10]; (b) the original dataset from Asadollah et Al. [11];(c) The data extracted by Arcan[12]; (d) the mapping and overlap between Arcan detections and dataset, with inter-rater reliability calculations[13].

A replication of this study is planned to strengthen the general validity of the conclusions perhaps over a more extensive dataset based solely on quantitative mining-software-repositories analysis.

5.4 CONCLUSIONS

This chapter outlined three different evaluations of Arcan:

- The evaluation of Arcan results on 7 open source projects and a preliminary comparison with the results of other two tools (e,g, inFu-

9 https://cran.r-project.org/package=irr
10 https://goo.gl/DhwHPq
11 https://goo.gl/DFg5f5
12 https://goo.gl/Jw51ND
13 https://goo.gl/TLiYpw

sion, Hotspot). The work has been published at ICSME 2016 [17], (Section 5.1).

- The evaluation of Arcan results carried out with real-life software developers in a industrial case study. Two different projects were analyzed by Arcan and a precision of 100% was observed, since Arcan found only correct instances of architectural smells. The work has been published at ICSA 2017 [19], (Section 5.2).
- The evaluation of the tool results through extensive mixed-methods research, that shows that the tool is *precise*, since it detects 100% of the architecture smells effectively present in the analyzed source code, and is reliable, since it detects the *correct* architecture smell in over 84% of the cases. This work has been submitted at the Journal of System and Software [135] (Section 5.3).

Moreover, another validation of Arcan results has been recently done in Sweden, in collaboration with the University of Gothenburg and some local companies. The results collected in this extended study through the software developers feedback will be evaluated to check if the architectural smells detected by Arcan are really perceived as problems and to get another overall usefulness evaluation of the tool.

6 EMPIRICAL ANALYSIS WITH ARCAN

This chapter describes some empirical analysis done through Arcan: the work on AS prediction and evolution considering also code changes in Section 6.1 and the evaluation of different architectural smells and their correlation with code smells in Section 6.2 in collaboration with the Free University of Bozen-Bolzano (Italy).

6.1 ARCHITECTURAL SMELL PREDICTION AND EVOLUTION

Architecture evaluation and quality assessment are important in order to identify problems that can lead to a progressive architecture degradation or erosion [102]. A prominent example of this kind of problems is given by architectural smells.

The identification of architectural smells is important, but it could be even more important to predict them before they actually appear. Moreover, if good predictors for the presence of architectural smells are found, then their causes could be better understood and their creation could be prevented.

A prediction approach needs to take into account different features (predictors) in the software history of the projects, which may include, among others, the presence of architectural smells, the number of involved developers, and the number of changes done in the project. If we consider the presence of architectural smells, we can observe that during the evolution of a project some architectural smells could be removed, some remain, or new ones can be introduced. Those that remain could be more critical to remove, as they can grow in size and complexity, or can co-evolve with other architectural smells during the evolution of the project.

The prediction of architectural smells can help software engineers during the typical development and maintenance activities and also during a continuous architecting process, through a constant exploration of some characteristics, such as the presence of architectural smells. This exploration can be implemented as a recommender system, warning against the risk of incurring an architectural smell.

In this study Arcan is used for AS detection on four open source projects. For each project, almost all the revisions found on their respective version control repository were considered.

The historical data on each revision of the projects is used, with the aim to predict the presence of architectural smells in future versions of the projects. The data collected are related to:

- three architectural smells detected without analyzing the history of the project: *Unstable Dependency*, *Cyclic Dependency*, *Hub-like Dependency* smells;

- one architectural smell where the detection is based on the history of the project: *Implicit Cross Package Dependency* smell;
- the number of developers which worked on the project;
- the number of changes at package level in terms of added or deleted parts of code.

With this objective, four different supervised learning models with the performance measures were applied in a repeated cross-validation setting .

Through this study, I aim to answer the following Research Questions (RQ):

- *RQ1* Can we use the presence of architectural smells in the project's history to predict the presence of architectural smells in the future?
- *RQ2* Can we use the number of the developers involved in a project or/and the number of changes done at package level in a project to predict the presence of architectural smells in the future?
- *RQ3* Can we use all the data on changes, developers and architectural smells to predict architectural smells?
- *RQ4* How does the number of architectural smells evolve in the project history? Which architectural smell tend to increase/decrease?

The answer to research question RQ1 explains whether there is a relation between the presence of architectural smells in past versions of the software and the future versions. If the relation can be understood by learning models with high performance, the models can be used to suggest parts of a project where architectural smells are going to be created, allowing architects to prevent this. Moreover, if machine learning models can exploit this information to make predictions, this means that some patterns in the presence of architectural smells exist, and are able to explain the presence of architectural smells in the future. This can be an important information, leading to future research dedicated to understandingthe co-evolution of architectural smells, i.e., the way they appear and influence each other along the history of a software project.

The answer to research question RQ2 aims to assess if the number of the developers involved in a project has an impact on the presence of architectural smells. The same evaluation has been done with respect to the number of changes computed by Git. If none of these two factors has an impact, the study will consider whether both the factors (information on the number of developers and the number of changes) in combination can act as predictors. Moreover, the answer to research question RQ3 will also analyze if the three factors in combination (information on the number of architectural smells, number of developers and number of changes) can act as predictors. The intention is to provide machine learning models and techniques based on the three factors to predict earlier if the project will be prone to be affected by new smells in the future according to how the project has been developed.

The answer to research question RQ4 analyze whether the number of architectural smells tends to increase or decrease over the project history. Moreover, the evolution of smells is studied to understand if they display some regularities in their evolution, e.g., if the number of their instances mostly increases or decreases over time. Knowing that an architectural smell tends to increase can help to focus the attention of developers on this smell and potentially plan possible refactoring actions to remove the smell.

6.1.1 *Definition and setup of the case study*

The case study aims is understanding if the architectural smells existing in the history of a project and the number of developers and number of changes can be used to predict the presence of architectural smells in future versions. In the following, I provide a description of the analyzed projects, how the information about the detected AS in the projects evolution are modelled, the machine learning algorithms applied, and the procedure applied to evaluate the learning performances.

6.1.1.1 *Selected projects*

Four projects are chosen from Github[1]: JGit, JUnit, Commons-Math and Apache Tomcat. The first three are libraries and the last is an application server. These projects are chosen since they are written in Java and have more than 5 years long activity, at least 2K commits, at least 20 major releases and a number of contributors higher than 10.

For JGit, all versions available in the `master` branch are considered, for a total of 83 versions (4840 commits in 8 years) and 138 contributors.

For JUnit, all versions available in the `master` branch are considered, for a total of 20 versions (2187 commits in 17 years) and 138 contributors.

For Commons-Math, all versions from the `master` branch are considered, for a total of 65 versions (6286 commits and 15 years) and 26 contributors.

For Tomcat, all versions from the project inception are considered (version 4, then named version 6) , up to the end of the development of version 7, actually analyzing two full major releases (v6 and v7), for a total of 127 versions (6853 commits in 6 years) and 13 contributors.

6.1.1.2 *Predictors and class labels*

In this study, AS evolution, number of developers and changes are used to predict the four AS. Since all AS are defined at package level, but they are not all defined at class level, it was chosen to work at package level, hence each data point in the dataset will refer to a package. In particular, since the detected AS rely on many versions of a project, a data point is given for each (version, package) combination. For each data point, AS are represented as follows:

1 https://github.com/

- `CD (Cyclic Dependency)`: binary feature, with value 1 if the package is involved in a cyclic dependency in that version, 0 otherwise;
  - `tiny, star, chain, circle, clique`: binary features, with value 1 if the package is involved in a cyclic dependency of the respective type in that version, 0 otherwise;

The same representation is used for the other three AS, i.e., `UD (Unstable Dependency)`, `HL (Hub-Like Dependency)` and `ICPD (Implicit Cross Package Dependency)`.

To predict AS, other data on the development history is considered, as the authors of the commits and some metrics on file changes extracted from the version control system (Git):

- *Author name*: the name (or nickname) used by the author to submit the commit,
- *Author email*: to identify exactly the author;
- *Number of files changed*: number of files changed in the commit for any type (Java files excluded), e.g., configuration file, read-me file;
- *Number of Java files changed*: number of Java files changed in the commit;
- *Addition-changes*: number of lines added in all the changed files (Java files excluded);
- *Deleting-changes*: number of lines removed in all the changed files (Java files excluded);
- *Addition-changes in Java files*: number of lines added in Java files;
- *Deleting-changes on Java files*: number of lines removed in Java files.

All the above features are used as predictors in the machine learning models, while AS features are used only as class labels, i.e., the target of the predictions. More precisely, predictors are used in past versions to predict labels in the next version.

6.1.1.3 *Representing architectural smells in history*

To be able to predict AS in future versions using the AS detected in past versions, a suitable representation is needed, enabling the application of prediction models on these data.

Lags are chosen as method to represent past data, i.e., past data are associated to current data by adding features to the same data point to represent both current data and the value of the predictors in the past at different times. For example, when dealing with project packages, for each package in each version there will be some features representing AS. In order to use the past 12 versions ("lags") to predict the presence of AS in the next version, the AS features of (package, version - 1), (package, version - 2), ...,

(package, version - 12) will be added to each row (package, version). In this way, every data point contains AS information about the current version and the 12 previous versions of the same package. Since this procedure is applied to all available versions, lags create a "sliding window" where for each version the information regarding a fixed amount of past versions is represented also. In the training phase, learning models use lagged information to model the current information. In the test phase, when using the models to predict new AS, they will use as input the current and lagged data. This representation is common when dealing with multivariate time series and has the advantage that, after this pre-processing phase, where lags are created, the remaining part of the experiment can be carried out as a standard supervised learning task.

The study started from a representation of the AS and development information (changes and developers) on each version (commit) of the analyzed project. Then, three variants of the dataset are generated aggregating by month. In the first one, the data associated to AS are used, in the second one the data on AS, number of changes and number of developers (development data), and in the third one only development data on changes and developers. These representations have different properties. First, by aggregating data by month synthesize information is possible and look at a large timespan, i.e., more information in the past to predict further information in the future. Second, the size of the datasets was reduced and hence the time needed to perform the analysis was reduced by a large extent. Third, whether the different sources of data (AS, changes and developers) had different relevance in the AS prediction could be understood.

6.1.1.4 *Machine learning models*

The following machine learning models were selected for the experiment, using R implementations:

- Naïve Bayes[2] (NB): Computes the conditional a-posteriori probabilities of a categorical class variable given independent predictor variables using the Bayes rule.
- Decision Trees[3] (C5.0): Decision tree learning uses a decision tree as a predictive model which maps observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves).
- Random Forests[4] (RF): A random forest is a classifier consisting of a collection of tree-structured classifiers $h(x, \Theta_k)$, $k = 1, ...$ where Θ_k are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input x [91].

2 https://cran.r-project.org/package=klaR
3 https://cran.r-project.org/package=C50
4 https://cran.r-project.org/package=randomForest

- Support Vector Machines[5] (SVMs, also support vector networks [41]) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. We tested two particular types provided by the R package: *svmRadial* and *svmLinear*.

6.1.1.5 *Performance estimation*

For the estimation of the performance of the models, a standard repeated k-fold cross validation procedure was applied, with 10 repetitions and 10 folds. Therefore, each model is evaluated 100 times on different portions of the dataset.

The `caret`[6] R package was used to perform all pre-processing, learning and performance evaluation tasks.

The computed performance indexes are the usual indexes found in machine learning and information retrieval in the case of supervised learning: Accuracy and F-Measure. These indexes are considered as 'positive' if the AS is present on a package, i.e., its respective feature is 1. Two pre-processing transformations are applied to the dataset before using it for learning. First, features with variance near to zero are removed (using the `nearZeroVar()` function in `caret`), mostly addressing columns having only one constant value. Then SMOTE [31] sampling is applied to balance the positive and negative class. Both operations are carried out with the default settings of the `caret` package.

To support an informed comparison of the different machine learning models performances, we apply the Wilcoxon signed rank test [62] to the F-Measure values obtained on the 10 folds of the cross validation procedure. Only performances of different models on the same dataset (system and pre-processing setup) are compared. Since all the models of each group are compared, a p-value correction is applied, using Holm's method [63]. In the discussion, the test results[7] are used to understand if the difference in the average F1 is significant, i.e., if the models' performances are distinguishable. The considered significance is $\alpha < 0.05$.

6.1.2 *Results*

In this section, the results obtained through this case study are discussed and explained, and the machine learning models are analyzed to understand which features have more influence on the prediction performances.

Table 6.1 reports computed F-measure and Accuracy metrics of all the models experimented on the four projects: the values higher or equal to 0.6 are highlighted in bold. The table reports prediction performances of five models for the next-*month* setup: the preceding 12 lags are used to predict the next one, e.g., the feature of the last 12 month are used to predict the

5 https://cran.r-project.org/package=kernlab
6 https://cran.r-project.org/package=caret
7 https://goo.gl/x9u5mZ

Table 6.1: F-measure and Accuracy results of machine learning models

			AS					C-D					AS-C-D				
			C5.0	NB	RF	SL	SR	C5.0	NB	RF	SL	SR	C5.0	NB	RF	SL	SR
Accuracy	Commons-Math	CD	**.995**	**.981**	**.995**	**.858**	**.823**	**.834**	**.828**	**.849**	**.858**	**.832**	**.995**	**.944**	**.995**	**.833**	**.823**
		CD-chain	**.837**	**.624**	**.831**	**.765**	**.732**	**.614**	.530	**.635**	**.765**	**.719**	**.844**	.592	**.856**	**.752**	**.732**
		CD-clique	**.945**	**.764**	**.942**	**.928**	**.920**	**.866**	.523	**.855**	**.928**	**.930**	**.958**	.577	**.954**	**.909**	**.920**
		CD-star	**.897**	**.610**	**.894**	**.922**	**.922**	**.898**	.427	**.877**	**.922**	**.922**	**.937**	.452	**.921**	**.912**	**.922**
		CD-tiny	**.845**	**.666**	**.846**	**.665**	**.676**	**.681**	**.618**	**.700**	**.665**	**.649**	**.854**	**.635**	**.841**	**.692**	**.677**
		UD	**.669**	.504	.597	.571	**.699**	**.651**	**.672**	**.663**	.571	**.627**	**.639**	**.674**	**.642**	**.800**	**.699**
		ICPD	**.711**	**.768**	**.712**	**.887**	**.862**	**.847**	**.869**	**.883**	**.887**	**.862**	**.845**	**.875**	**.873**	**.857**	**.862**
	JGit	CD	**.996**	**.986**	**.996**	**1**	**1**	**.774**	.482	**.860**	**1**	**.977**	**.995**	**.956**	**.994**	**1**	**.965**
		CD-chain	**.833**	**.665**	**.846**	**1**	**.993**	**.679**	**.658**	**.683**	**1**	**.902**	**.813**	**.659**	**.818**	**1**	**.856**
		CD-circle	**.765**	**.630**	**.782**	**1**	**.995**	.573	.496	.566	**1**	**.914**	**.753**	.506	**.761**	**1**	**.871**
		CD-clique	**.953**	**.799**	**.955**	**1**	**.997**	**.798**	**.846**	**.790**	**1**	**.923**	**.918**	**.847**	**.917**	**1**	**.894**
		CD-star	**.785**	**.646**	**.800**	**1**	**.994**	**.665**	**.638**	**.662**	**1**	**.886**	**.782**	**.643**	**.779**	**1**	**.842**
		HL	**.684**	**.747**	**.753**	**1**	**.985**	**.826**	**.880**	**.810**	**1**	**.957**	**.836**	**.882**	**.821**	**1**	**.955**
		UD	.585	**.638**	**.732**	**1**	**.990**	**.828**	**.851**	**.831**	**1**	**.942**	**.840**	**.855**	**.843**	**1**	**.922**
		ICPD	**.671**	.590	**.710**	**1**	**.981**	**.802**	**.808**	**.809**	**1**	**.891**	**.787**	**.807**	**.803**	**1**	**.852**
	JUnit	CD	**.998**	**.896**	**.999**	**1**	**.958**	**.924**	**.671**	**.907**	**1**	**.958**	**.998**	**.820**	**.999**	**1**	**.919**
		CD-chain	**.998**	**.899**	**.998**	**1**	**.847**	**.616**	.551	**.636**	**1**	**.847**	**.999**	**.836**	**.998**	**1**	**.752**
		CD-star	**.994**	**.753**	**.993**	**1**	**.903**	**.792**	**.679**	**.804**	**1**	**.903**	**.995**	**.720**	**.995**	**1**	**.863**
		CD-tiny	**.997**	**.884**	**.997**	**1**	**.857**	**.665**	.504	**.649**	**1**	**.857**	**.997**	**.855**	**.997**	**1**	**.786**
		UD	**.939**	.587	**.937**	**1**	**.823**	**.709**	**.671**	**.742**	**1**	**.823**	**.957**	.575	**.959**	**1**	**.758**
		ICPD	**.997**	**.959**	**.997**	**1**	**.874**	**.992**	**.917**	**.991**	**1**	**.874**	**.998**	**.942**	**.999**	**1**	**.792**
	Tomcat	CD	**.999**	**.977**	**.999**	**1**	**.999**	**.791**	.280	**.762**	**1**	**.952**	**.999**	**.899**	**.997**	**1**	**.955**
		CD-chain	**.996**	**.913**	**.997**	**1**	**.998**	**.718**	.475	**.723**	**1**	**.940**	**.996**	**.832**	**.996**	**1**	**.917**
		CD-clique	**.996**	**.793**	**.996**	**1**	**.999**	**.787**	**.856**	**.820**	**1**	**.973**	**.996**	**.927**	**.996**	**1**	**.939**
		CD-star	**.989**	**.760**	**.986**	**1**	**.995**	**.753**	.518	**.726**	**1**	**.940**	**.992**	.581	**.990**	**1**	**.914**
		ICPD	**.750**	**.620**	**.723**	**1**	**.996**	**.808**	**.883**	**.827**	**1**	**.951**	**.834**	**.884**	**.836**	**1**	**.936**

			AS					C-D					AS-C-D				
			C5.0	NB	RF	SL	SR	C5.0	NB	RF	SL	SR	C5.0	NB	RF	SL	SR
F-measure	Commons-Math	CD	**.997**	**.989**	**.997**	.537	.421	**.902**	**.906**	**.912**	.537	.449	**.997**	**.966**	**.997**	.453	.421
		CD-chain	**.788**	**.643**	**.783**	**.804**	**.776**	.490	.479	.479	**.804**	**.766**	**.810**	.494	**.823**	**.793**	**.777**
		CD-clique	**.767**	.384	**.762**	**.960**	**.956**	.387	.213	.310	**.960**	**.961**	**.795**	.239	**.782**	**.950**	**.956**
		CD-star	.553	.269	.557	**.958**	**.958**	.314	.180	.325	**.958**	**.958**	**.610**	.187	.550	**.953**	**.958**
		CD-tiny	**.816**	**.694**	**.815**	**.706**	**.716**	**.631**	.360	**.641**	**.706**	**.692**	**.835**	.388	**.825**	**.730**	**.717**
		UD	.039	.419	.221	**.685**	**.779**	.391	.217	.368	**.685**	**.726**	.348	.216	.342	**.853**	**.779**
		ICPD	.235	.211	.255	**.938**	**.924**	.176	.076	.178	**.938**	**.924**	.258	.070	.267	**.921**	**.924**
	JGit	CD	**.998**	**.992**	**.998**	**1**	**1**	**.870**	**.615**	**.924**	**1**	**.987**	**.997**	**.975**	**.997**	**1**	**.981**
		CD-chain	**.757**	.437	**.775**	**1**	**.990**	.442	.028	.444	**1**	**.826**	**.728**	.028	**.741**	**1**	**.732**
		CD-circle	**.756**	**.713**	**.769**	**1**	**.994**	.508	**.640**	.505	**1**	**.900**	**.751**	**.648**	**.752**	**1**	**.845**
		CD-clique	**.849**	.500	**.855**	**1**	**.990**	.316	.011	.369	**1**	**.663**	**.749**	.012	**.751**	**1**	.479
		CD-star	**.683**	.443	**.726**	**1**	**.992**	.411	.070	.437	**1**	**.816**	**.685**	.094	**.692**	**1**	**.725**
		HL	.175	.211	.121	**1**	**.827**	.147	.172	.151	**1**	.489	.230	.178	.220	**1**	.248
		UD	.233	.255	.132	**1**	**.959**	.340	.160	.302	**1**	**.671**	.386	.176	.334	**1**	.529
		ICPD	.252	.305	.232	**1**	**.950**	.499	.225	.506	**1**	**.622**	.491	.220	.506	**1**	.427
	JUnit	CD	**.998**	**.871**	**.999**	**1**	**.977**	**.957**	**.777**	**.946**	**1**	**.977**	**.998**	**.753**	**.999**	**1**	**.956**
		CD-chain	**.998**	**.884**	**.998**	**1**	**.875**	**.663**	.375	**.678**	**1**	**.875**	**.999**	**.804**	**.998**	**1**	**.812**
		CD-star	**.995**	**.793**	**.993**	**1**	**.622**	.370	.285	.351	**1**	**.622**	**.996**	**.753**	**.996**	**1**	.369
		CD-tiny	**.997**	**.893**	**.997**	**1**	**.724**	.403	.537	.537	**1**	**.724**	**.998**	**.869**	**.997**	**1**	.519
		UD	**.943**	**.715**	**.941**	**1**	.524	.351	.416	.332	**1**	.524	**.960**	**.697**	**.962**	**1**	.207
		ICPD	**.996**	**.954**	**.997**	**1**	**.898**	**.993**	**.922**	**.992**	**1**	**.898**	**.998**	**.933**	**.998**	**1**	**.844**
	Tomcat	CD	**.999**	**.986**	**.999**	**1**	**.999**	**.876**	.261	**.849**	**1**	**.972**	**.999**	**.937**	**.998**	**1**	**.973**
		CD-chain	**.997**	**.933**	**.998**	**1**	**.999**	**.790**	.432	**.794**	**1**	**.957**	**.997**	**.867**	**.997**	**1**	**.941**
		CD-clique	**.982**	.409	**.982**	**1**	**.995**	.139	.096	.138	**1**	**.834**	**.977**	.567	**.979**	**1**	.559
		CD-star	**.98**	**.666**	**.976**	**1**	**.990**	.497	.479	.544	**1**	**.882**	**.985**	.513	**.983**	**1**	**.820**
		ICPD	.292	.269	.289	**1**	**.978**	.268	.031	.195	**1**	**.668**	.383	.034	.337	**1**	.541

Legend: Architectural Smell (AS); Changes and Developers only (C-D); Architectural Smell, Changes and Developers data (AS-C-D); SVM linear (SL); SVM Radial (SR); Decision Tree (C5.0); Naïve Bayes (NB); Random Forest (RF)

In **bold** are highlighted values equals or greater to 0.6

AS in the next month. The models used are the decision tree (C5.0), Naïve Bayes (NB), Random Forest (RF), Support Vector Machine Linear (SL) and Support Vector Machine Radial (SR).

The first column of the table reports the different AS, where for CD smell we have considered all the different shapes and the sum of all the CD (CD

row), the second column reports AS prediction using only AS data (*AS*) to answer RQ1, the third column reports AS prediction using only Changes and Developers data (*C-D*) to answer RQ2 and the forth column reports the AS prediction using AS, Changes and Developers data (*AS-C-D*) to answer RQ2.1.

The reported performances are high with very few exceptions. The highest performances are obtained in the prediction of Cyclic Dependencies (CD) smell and its shapes. In fact, only CD has comparable detection performances in both *AS* and *AS-C-D* setups. This result could also be influenced by the class imbalance: accuracy values for *NB* predictions are much lower than the others (and higher than F-Measure) for all the projects. This means that where class imbalance is higher, classifiers chose the largest class (absence of AS). As for the classifiers, the best ones are SVM Linear and SVM Radial: this conclusion is supported by the Wilcox test conducted and explained in Section 6.1.1.5. The test shows that p-values of SL and SR models are significant (p-values between 0 and 0.03), with respect to all models using C-D data and AS-C-D data; this is true for the majority of the test on AS data too. C5.0 and Random Forests are also good, and have comparable performances among each other.

Rules computed by the JRip algorithm available in Weka are analyzed and are accessed through R using the RWeka[8] package, because I experienced that the rules extracted by JRip are more understandable than the ones reported by C5.0, while the latter can have higher prediction performances. the rules were used to investigate which conditions can lead to the creation of architectural smells. the training dataset is simplified, representing only the presence or absence of architectural smells in lagged data (0 absence and 1 presence).

the main high-level rule discovered is that the presence (or absence) of an architectural smell in the past is confirmed in the future. This can be justified by the fact that, for their granularity, architectural smells are large and can have a slow evolution.

6.1.2.1 *Results on architectural smells prediction (RQ1)*

We analyzed if we can use the presence of architectural smells in the history of the projects to predict the presence of AS in the future.

As shown in Table 6.1, SVM models performed well using this data, hence AS evolution could be used for the prediction of AS in the future. We collected and reported the most important rules extracted using this type of data. Table 6.1 shows that the presence of Cyclic Dependencies indicates a possible presence in the future of this architectural smell, and this holds also for the absence: If the cycle is not present in the history it will not be present in the future. This is true for all the systems and for all the shape types of Cyclic Dependencies.

8 https://cran.r-project.org/package=RWeka

For example, the following rule is extracted from the analysis of JGit (with 97.6% correctly classified instances):

$$(CD1 = 0) \Rightarrow CD = 0 \tag{6.1}$$

Table 6.2: Prediction performance rule of CD Smell

	Instances classified by $(CD1 = 0) \Rightarrow CD = 0$			Total number
Project	**Right**	**Wrong**	**Right(%)**	**of instances**
Commons Math	516	14	97.4%	3288
JGit	332	8	97.6%	2989
JUnit	690	20	97.2%	6057
Tomcat	791	0	100%	5306

While the rule does not indicate the way CD is introduced or removed, it shows that the absence of CD in the last month indicates its absence in the future.

What is found in JGit is confirmed also for all the other projects and for all the shapes of CD. Table 6.2 shows the rule performance for all projects, and it also indicates the number of total instances of CD per project. Tomcat is the only project where the rule is always true, but the other projects achieved good results with correct classification percentage over 97%. Since the rule does not describe a condition that leads to the introduction (or removal) of the CD AS, this rule as a witness of the sparsity of the phenomenon. It also has some practical implications: since the introduction of CD is rare, paying attention in the early phases of the construction of a system or module could decrease the possibility of incurring CD during evolution. Obviously, since the rule does not have 100% confidence on all projects, developers have to proactively avoid the introduction of CD, but the event has low chances to happen. As future work, it will be interesting to analyze the reasons why this AS has been introduced in the analyzed projects, i.e., the cases where the rule fails. Moreover, I found that if in two points in the past there is no Chains or Tiny CD smell, then in the future a Star shaped CD smell will be introduced.

For what concerns Hub-Like smell, Unstable Dependency and Implicit Cross Package Dependency smell, their rules were not extracted or not significantly relevant.

6.1.2.2 *Results on architectural smells prediction (RQ2 and RQ3)*

The analysis investigates if the number of the developers involved in a project or/and the number of changes done at package level in the project can be used to predict the presence of architectural smells in the future versions of the project (according to RQ2). No significant results were obtained by individually considering as indicator first the developers involved and

then the number of changes. When considering both of them as shown in Table 6.1, SVMs models gave the best results, even if it was not possible to extract any interesting rules. Instead, prediction rules were obtained using together data on changes, developers and architectural smells as predictors (according to RQ3).

Table 6.3: Prediction rules leading to addition of UD Smell

Id	Parameter	Month window 12	11	10	9	8	7	6	5	4	3	2	1	(R/W)
1	CD-tiny										$= 0$			JUnit
	CD-chain	≥ 1												70/24
	JFC										≥ 1	≥ 1	≥ 1	
2	CD-tiny											$= 0$		JUnit
	CD-chain	≥ 1												48/18
	JFC	≥ 1	≥ 1	≥ 1										
3	CD-tiny												$= 0$	JUnit
	CD							≥ 1						36/6
	JFC						≥ 1	≥ 1	≥ 1		≥ 1		≥ 1	
4	CD-tiny											$= 0$		JUnit
	CD							≥ 1						144/71
	JFC					≥ 1		≥ 1						
5	A-C			≥ 1										JGit
	D-CJF			$= 0$			$= 0$							25/6
	FC						≥ 1							
6	A-C	≥ 1												JGit
	D-CJF	$= 0$												
	FC								≥ 1					39/15
	D-C						$= 0$							
7	A-C	≥ 1												JGit
	D-CJF					$= 0$								
	FC		≥ 1		≥ 1									18/5
	D-C					≥ 1			$= 0$					

Legend: Java File Changed (JFC); Files Changed (FC); Deleting-Changes (D-C); Addition-Changes (A-C); Right/Wrong Classified item (R/W);Deleting-Changes in Java File (D-CJF)

Tables 6.3 and 6.4 show respectively the 7 prediction rules of UD smell and the 6 prediction rules of ICPD smell. A rule is identified by an *id* (column 1) and it is composed by some parameters joined by logical *AND* (column 2). A time-window of 12 lags, i.e., 12 months before are given to the machine learning algorithm. If a value is important for the rule, it is reported in the proper n-month column, otherwise the column is left empty. The last column reports the performances of the analyzed project by showing the number of Right and Wrong (R/W) instances classified by the rule and the name of the analyzed project.

As shown in Table 6.3 for UD smell prediction, rules (1-4) are valid only for JUnit and rules (5-7) are valid only for JGit. All the rules regarding JUnit

indicate that an UD smell is introduced if there is no instance of CD-tiny in one of the last quarter months. This happens in conjunction with the other parameters: in particular, rule 1 and rule 2 indicate that one or more CD-chain smells should be present twelve months before; moreover, one or more Java files should be changed in the last three months as in rule 1 or one year before as in rule 2. Regarding rule 3 and rule 4, Table 6.3 tells us that Java file changes made in a period of 5 months or a shorter period of 2 months with the presence of one or more CD smell in the past six months can lead to the introduction of an UD smell.

Rules (5-7) show that a new UD smell can be introduced if in the past there were not deleting changes to Java files but addition changes and file changes to other files (not Java). However, rules (5-7) are valid only for JGit and in few other cases.

Table 6.4: Prediction rules leading to addition of ICPD Smell

Id	Parameter	Month window 12	11	10	9	8	7	6	5	4	3	2	1	(R/W)
1	D-CJF	≥ 1											≥ 1	JGit
	D-C											≥ 1		
	CD						$= 0$							45/1
	FC										≥ 1			
2	D-CJF				≥ 1			≥ 1					≥ 1	JGit
	D-C											≥ 1		
	FC			≥ 1										219/102
3	D-CJF				≥ 1		≥ 1	≥ 1						JGit
	A-C		≥ 1											113/55
	A-CJF										≥ 1			
4	CD-Chain	$= 0$									$= 0$			Tomcat
	FC						≥ 1				≥ 1			
	A-C												≥ 1	38/15
	D-C											≥ 1		
5	CD-Chain	$= 0$				$= 0$								Tomcat
	FC			≥ 1						≥ 1				36/17
	CD									$= 0$				
6	CD-Chain	$= 0$											$= 0$	Tomcat
	FC		≥ 1											
	A-C					≥ 1								18/4
	D-C	$= 0$										≥ 1		

Legend: Deleting-Changes in Java File (D-CJF); Deleting-Changes (D-C); Number of Files Changed (FC); Addition-Changes (A-C); Addition-Changes in Java File (A-CJF)

Table 6.4 shows the rules valid for the ICPD smell. Deleting changes made in Java file are important for the introduction of an ICPD smell, since deleting changes are involved in all the rules reported in Table 6.4. Deleting changes in other files are also significant for the introduction of ICPD as

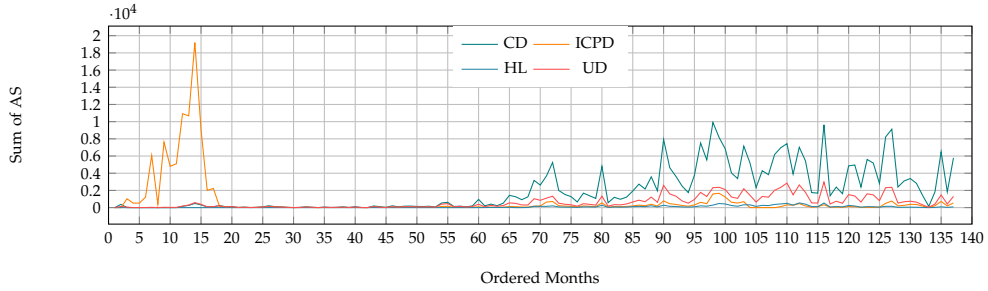


Figure 6.1: Evolution of architectural smells of Commons-Math by month

shown in rules (1-2) which show the importance of modifying one or more files in the same rules. Rule 3 shows the relevance of additional changes both in Java file and in a general file. Rule 1 shows that CD smell, 7 month before should be zero. Moreover, Tomcat rules showed that CD-chain should be zero in at least two points in the past, as shown in rules (4-6).

CD smell rule indicates that if there is no CD smell in the past month there will none in the next month. This rule is still valid with similar performances in all projects and the rule was shown also using the AS dataset, as previously explained in RQ1 answer in Section 6.1.2.1.

Concerning the prediction of the other AS, changes data are good indicators for the insertion of new shapes of CD smells in the project. In fact, we noticed that adding lines to a Java file without removing them for a long period could lead to a new shape of CD, in particular: chain and circle.

HL smell did not reveal any hidden rule with other architectural smell, i.e., there is no type of rule which concerns the introduction or removal of HL smell.

Finally, as described before, changes and architectural smell data are good predictors of AS, in contrast with developers data which are not useful to predict AS.

6.1.2.3 *Results on architectural smells evolution (RQ4)*

The evolution of the number of the AS detected in the history of the analyzed projects is analyzed to observe if AS tend to increase or decrease. If a specific instance of an AS remains or is removed is not considered, but their overall number. Figures 6.1-6.4 show the evolution of the four AS for each project. The majority of smells shown that are of type CD and ICPD during the evolution of all projects. Hub-Like smells are usually the most rarely detected.

These figures visually indicate the presence of a correlation among architectural smells quantity during the evolution of the projects, i.e., architectural smells increase and decrease together. The correlation is more evident in Figure 6.2 where there are peaks of the plot aligned over different architectural smells, but it is evident also in the other projects.

The evolution of architectural smells in Commons-Math is shown in Figure 6.1, and an increasing trend of ICPD is shown for the first period of

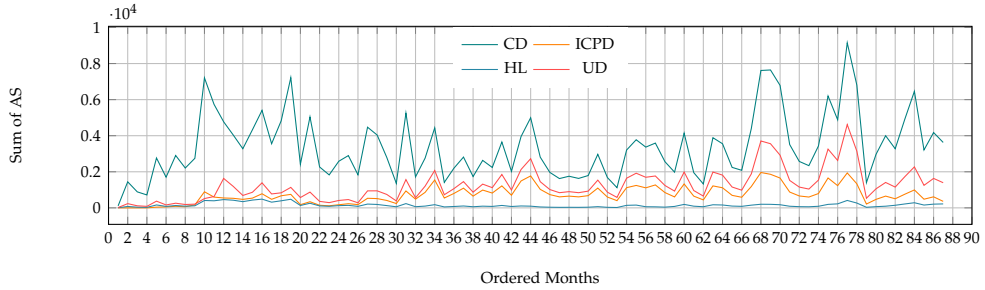


Figure 6.2: Evolution of architectural smells of JGit by month

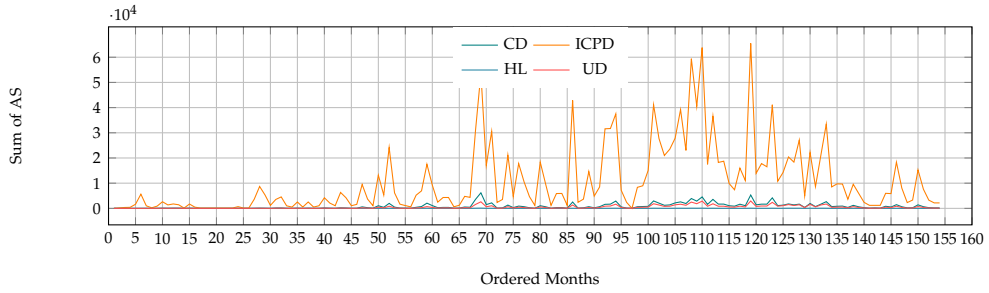


Figure 6.3: Evolution of architectural smells of JUnit by month

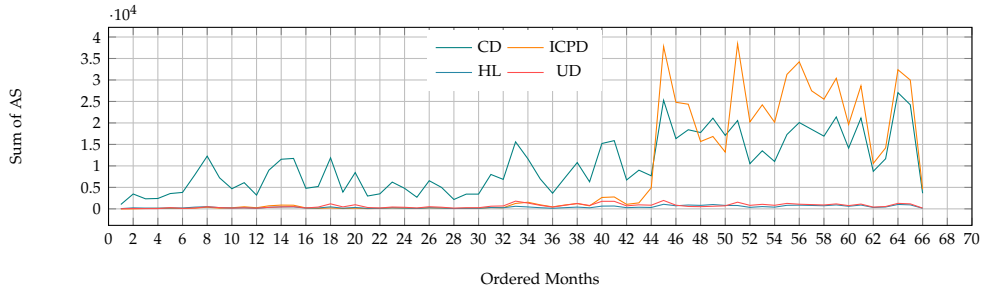


Figure 6.4: Evolution of architectural smells of Tomcat by month

20 months, and a substantial growth trend for all the architectural smells during the entire considered period.

Figure 6.2, related to JGit project, shows that CD is much more present than the other types of architectural smells and there is a higher number of UD w.r.t. ICPD and HL.

Figure 6.3 shows architectural smells evolution in JUnit and an ICPD prevalence is shown with the highest value of 60K instances of this smell with respect to the other projects and the other smells. In fact, the other types of architectural smells are less than 1K instances and less than the other projects since JUnit is the smallest analyzed project. However, this project presents a lot of ICPD smells because developers modify more than one file per commit: this was revealed thanks to a manual check on a sample of commits. The interpretation of this result is that JUnit is a project developed basically by its original creators who know exactly how the project is structured. They do commit by changing a lot of files since their deep knowledge of the project allows them to easily spot and fix problems also among unrelated files.

Tomcat architectural smells evolution is shown in Figure 6.4. ICPD smells are increasing over time and become the most detected smell after the 45th month, while the CD smells are the most detected ones until the 45th month.

6.1.3 *Threats to validity*

Threats to *construct validity* of this study may arise from the representativeness of the applied measures. This analysis exploited datasets where data points have been aggregated. In fact, we considered data representing the projects' characteristics on a monthly basis, but the raw extracted data refers to single commits. During the application of standard aggregation approaches, i.e., aggregating measures representing counts using the sum and aggregating ratios using their mean, the aggregation may have biased the dataset composition. In fact, since original data refers to single commits, it is not directly tied to a time interval (such as hours, days, and weeks). By aggregating the measures taken at irregular time intervals, distortions or "masking effects" may occur, e.g., when the same portion of code has been modified many times between two points of interests (months in this case), they are counted differently than the way they would have been count if the state of the repository was observed on a monthly basis. These issues are mitigated by choosing aggregation strategies keeping (as much as possible) the original meaning of each measure. Moreover, the study rely on a tool (Arcan) to extract dependency metrics and detect architectural smells in the analyzed projects. The tool could be subjected to a systematic bias in the detection, even if a validation of Arcan results has been performed [19]. Anyway, since the output of the tool is being predicted in the future, this systematic bias, if existing, has been incorporated into the learning models. Before using the data extracted by the tool to perform the experiment, the output was carefully checked, on many different projects, to optimize the settings of the detection rules and to verify the correctness of the tool output

according to the defined rules. The reproducibility of the experiment is guaranteed by the fact that the tool is freely available and can be applied to any compiled Java project. The reproducibility package of this study is available at this link[9].

Threats to *external validity* may arise from the the number of projects used in the experiment. The results obtained on the four projects may not be replicable on other systems, in particular if they do not rely on the same technology or belong to different domains. Another factor that may change the analysis, and any empirical study working on the evolution of software systems, is the set of practices applied by the team developing the analyzed projects. Still, machine learning can be a viable solution to apply customized prediction models which adapt on a project specific features, by relying on past data.

6.1.4 *Conclusions*

In this study, machine learning techniques are applied to predict architectural smells based on historical smell information. The evolution of the AS is analyzed based on historical information. My aim was to understand if the evolution of the presence of architectural smells in a project, and data related to the number of changes and the number of developers can be used to predict the existence of AS in the future. The ability of predicting AS can be used, for example, to understand if some parts of the project are subject to a risk of having one or more smells, enabling developers to prevent them. In the following, findings were summarized through the answers to the posed RQs.

With respect to RQ1: *Can we use the presence of architectural smells in history to predict the presence of AS in the future?* From the results obtained in the four analyzed projects, the prediction performances are high or very high. So, the answer to the RQ1 is *yes*, historical architectural smell information can be used to predict the presence of AS in the future. In particular, we found that Cyclic Dependency (CD) AS is a good predictor of CD smells. Hence, developers/maintainers have to pay particular attention to this AS and try to remove it, as soon as the AS is introduced and detected.

With respect to RQ2: *Can we use the number of the developers involved in a project or/and the number of changes done at package level in the project to predict the presence of architectural smells in the future versions of the project?*, The answer is *Yes* according to changes and *No* according to developers, since the rules detected by the models have not considered developers data and only SVM models get good F-measure values in average.

With respect to RQ3: *Can we use all the data on changes, developers and architectural smells to predict architectural smells?* As the second column of Table 6.1 shows, these indicators gave good Accuracy values. The changes on Java and not Java files, together with presence/absence of particular shapes of CD smell, indicate the presence of future UD smell; ICPD smell is announced by

9 https://goo.gl/MxHKGa

deleting/adding changes on both Java and not Java files, in conjunction with the absence of CD smell in the previous months. These results are shown in Tables 6.3-6.4 respectively. Changes are good indicators of insertion of new shapes of CD smells, in particular CD-chain and CD-circle smells. HL smell did not reveal any interesting rule. These results assessed that changes data and AS data are good predictors, in contrast with developers data.

With respect to RQ4: *How does the number of architectural smells evolve in the project history? Which architectural smell tend to increase/decrease?* The architectural smells are visually correlated and generally increase over time. There is a majority of CD smells and ICPD smells. ICPD smells usually have a different evolution compared to the other AS, but they have an increasing trend too. More ICPD instances were detected in the smallest project, JUnit; in fact, they are related not only to the size of the project but mainly to how the projects are developed. From the results on the detection of the ICPD AS, it is shown that the smell is present in all the four projects throughout their history and can have large variations according to the development style. Some false positives were detected and were related to some large batches of changes not directly related to development, e.g., changes in the software licence header of many files.

On this study a paper has been submitted to ICSME 2018 Conference [137].

6.2 CODE SMELLS AND ARCHITECTURAL SMELLS CORRELATIONS

Several works claim that architectural smells lead to architectural erosion and architectural issues are the greatest source of technical debt [47], hence they have to be considered as one of the primary source of investigation to face and mitigate the architecture degradation problem [118].

Several works, as outlined in Section 2.3.1 investigated inter-relations between code smells, if a code smell lead to another code smell, or if some code smells tend to go together. Currently, few studies analyzing correlations between code smells and architectural smells have been published. Among them, one work [102] identified correlation between code smells and architectural smells, while another work [101] claims that they are not correlated, but in any case, no extended empirical evaluations have been carried out and no code smell stands out as the best indicator of harmfulness respect to architecture degradation.

The goal of this study is to understand if architectural smells can be derived from a combination of code smells, or by a category of code smells[105]. For this purpose, we designed and conducted a large empirical study analyzing correlations on four architectural smells and 19 code smells and six categories of code smells on 111 Java project from the Qualitas Corpus repository [154].

The result of this work will support researchers and practitioners in understanding if they should detect both architectural smells and code smells or if code smells detection alone is enough to highlight the same anomalies that could be highlighted by the architectural smell. Results show that ar-

chitectural smells cannot be derived from code smells, and that in most of cases, code smells infect different classes than these infected by architectural smells, not only hindering different problems but also considering different candidate classes for refactoring.

6.2.1 *Background*

In this Section, we present the code smells together with their proposed classification and the architectural smells detected in this work.

6.2.1.1 *Code Smells*

In this work, we consider code smells detected by SonqaQube [10] using the "Antipatterns-CodeSmell" plugin [11]. All the code smells, except Duplicated Code are detected by the "Antipatterns-CodeSmell" plugin while Duplicated Code is detected natively by SonarQube. Here is the list of code smells considered in this work:

- *Anti Singleton (ASG)*: A class that is providing mutable class variables, which exhibit the properties of global variables [74].
- *Base Class Knows Derived Class (BCKD)*: A base class has knowledge about its derived class [116].
- *Base Class Should Be Abstract (BCSA)*: An inheritance tree contains roots that are not abstract - only the leaves should be concrete [116].
- *Blob (BL)*: Procedural-style design results in one object with numerous responsibilities more uniformly and responsibilities and most other objects holding only data [33].
- *Class Data Should Be Private (DsP)*: A class that publicly exposes its variables [151].
- *Complex Class (CC)*: A class with high MCCabe's cyclomatic complexity [107].
- *Duplicated Code (DC)*: A class or method that contains an identical piece of code of another class of method. Please, note that we only consider internal project duplication and we do not consider cross-project duplications.
- *Functional Decomposition (FD)*: Non object oriented design (possibly from legacy) is coded in object oriented language and notation [33].
- *Large Class (LC)*: A class with too many lines of code, methods or variables [48].

10 SonarQube https://www.sonarqube.org/
11 SonarQube https://github.com/davidetaibi/sonarqube-anti-patterns-code-smells

- *Lazy Class (LzC)*: A class that is doing very little work and only increases architectural complexity [48].
- *Long Method (LM)*: A method with too many lines of code [48].
- *Long Parameter List (LPL)*: A method having too many parameters [48].
- *Many Field Attributes But Not Complex (MFnC)*: A class that is not complex but has many public fields [116].
- *Message Chains (MC)*: A chain of methods that ask for one object, which asks for another one, which asks for yet another and so on [48].
- *Refused Parent Bequest (RPB)*: Subclass is using only a few features of the parent class [48].
- *Spaghetti Code (SC)*: An ad hoc software structure makes it difficult to extend and optimize code [33].
- *Speculative Generality (SG)*: Hooks and special cases in code, that handle things, which are not required, but are speculated to be required someday [48].
- *Swiss Army Knife (SAK)*: Over-design of interfaces results in objects with numerous methods that attempt to anticipate every possible need. This leads to designs that are difficult to comprehend, utilize, and debug, as well as implementation dependencies [33].
- *Tradition Breaker (TB)*: An inherited class provides a large set of new services that are unrelated to those provided by the base class [107].

6.2.1.2 *Categories of code smells*

The categories of code smells we have considered are based on the classification proposed by Mäntylä and Lassenius cited[105], where the smells are classified according to some of the common concepts that the smells inside one category share. Below we provide the description of each category with the included smells among those which we are able to detect with the Antipatterns-CodeSmell tool. Moreover, we outline the new smells that we included in the categories, if any.

- *The Bloaters (Bloat.)*: Objects that has grown too much and can become hard to manage. This group includes the code smells *Blob*, *Long Method*, *Large Class*, and *Long Parameter List*. We included: *Complex Class* and *Swiss Army Knyfe*.
- *The Dispensables (Disp.)*: Unnecessary code fragments that should be removed. It includes the Code Smells *Lazy Class*, *Duplicated Code* and*Speculative Generality*. We included: *Many Field Attributes But Not Complex* and *Duplicate Code*.

- *The Encapsulators (Enc.)*: Objects that presents high coupling (this category is also called *Couplers*). This group includes the code smells *Message Chain*.
- *The Object-Orientation Abusers (OOA)*: Classes that does not comply with the object-oriented design. For example, a Switch Statement, even if applicable in procedural programming, is highly deprecated in object-oriented programming. This group includes the code smells *Antisingleton*, *Refused Parent Bequest*. We included: *Base Class Knows Derived Class*, *Base Class Should Be Abstract*, *Class Data Should Be Private* and *Tradition Breaker*.
- *The Change Preventers*: This group includes smells that hinder further changes in the source code. This group includes a set of code smells such as *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance Hierarchies*, not detected by the Antipatterns-CodeSmell tool. We included: *Spaghetti Code*.

Moreover, other code smells detected by the Antipatterns-CodeSmell tool could be grouped together in a new category:

- *The Object-Oriented Avoiders* This category is in contrast to *The Object-Orientation Abusers*, since code smells belonging to this category are not applying (intentionally or not) any Object-oriented practice. We included the code smell *Functional Decomposition*.

Since three categories (*The Change Preventers*, *The Encapsulators*, *The Object-Orientation Avoiders* are based only on one code smell, we do not analyze them independently since they will provide the same results of the same code smell belonging to them. Therefore, in the remainder of this work we consider three categories of code smells: *The Bloaters*, *The Dispensables* and *The Object Orientation Abusers*.

In Table 6.5 we propose a summary of the new revisited classification of the smells with all the categories we have considered and the smells included in each category. In the table in *italic* we outline the new smells we have introduced in the categories of Mantylia according to our evaluation and the new category that we have defined.

6.2.1.3 *Architectural Smells*

The considered architectural smells of the study are the ones detected by Arcan (see Section 4):

- *Unstable Dependency (UD)*
- *Hub-Like Dependency (HD)*
- *Cyclic Dependency (CD)*
- *Multiple Architectural Smell (MAS)*.

Table 6.5: Code Smells Taxonomy

Category Name	Code Smells
The Bloaters	Blob
	Large Class
	Long Method
	Long Parameter List
	Complex Class
	Swiss Army Knife
The Change Preventers	*Spaghetti Code*
The Dispensables	Lazy Class
	Speculative Generality
	Many Field Attributes But Not Complex
	Duplicate Code
The Encapsulators	*Message Chain*
The Object-Orientation Abusers	Antisingleton
	Refused Parent Bequest
	Base Class Knows Derived Class
	Base Class Should Be Abstract
	Class Data Should Be Private
	Tradition Breaker
The Object-Orientation Avoiders	*Functional Decomposition*

We decided to consider these AS in the study, since they represent relevant problems related to dependency issues that lead to architectural degradation.

In the following section the case study design is described following the guideline proposed by Runeson [138].

6.2.2 *Case Study Design*

The goal of this work is to understand if architectural smells can be considered derivable and obtainable from code smells or if they are independent from them. For this purpose, we conducted a case study to investigate the interdependency between architectural smells and code smells analyzing 111 open source Java projects.

In this section, we present research questions, metrics and hypotheses for the case study. Based on them, we outline the study context, the data collection and the data analysis.

Based on our goal, we derived the following Research Questions (RQs):

RQ1 : Is the presence of an architectural smell independent from the presence of code smells?

RQ1.1 : Is the presence of a Multiple Architectural Smell (MAS) independent from the presence of code smells?

RQ2: Is the presence of an architectural smell independent from the presence of a *category* of code smells?

RQ2.1 : Is the presence of a Multiple Architectural Smell independent from the presence of a *category* of code smells?

With our RQs, we aim to understand if a single architectural smell (**RQ1**) or the multiple Architectural Smell (**RQ1.1**) can be calculated from code smells - or from a category that groups code smells as described in Section 6.2.3 (**RQ2** and **RQ2.1**).

Based on the aforementioned RQs, we defined the following hypotheses.

hypothesis 1 : The presence of an architectural smell is independent from the presence of code smells.

hypothesis 2 : The presence of a Multiple Architectural Smell is independent from the presence of code smells.

hypothesis 3 : The presence of an architectural smell is independent from the presence of a *category* of code smells.

hypothesis 4 : The presence of a Multiple Architectural Smell is independent from the presence of a *category* of code smells.

6.2.2.1 *Study Context*

The selected projects are contained in the Qualitas Corpus collection of software systems [156]. In particular, we used the compiled version of the Qualitas Corpus [154]. 111 Java projects are available and already compiled with more of 18 millions of LOCs, 16 thousands of package and 200 thousands classes analyzed. The dataset includes projects from different contexts such as IDEs, SDKs, Databases, 3D/graphics/media, diagram/visualization libraries and tools, games, middlewares, parsers/generators/make tools, programming language compilers, testing libraries and tools, other tools that do not belong to the previous categories. More information on the projects context and types can be found in [156]

6.2.3 *Data Collection*

The architectural smells are detected on 111 Java projects and code smells in 103 Java projects of the Qualitas Corpus [156], as depicted in Figure 6.5.

Architectural smells have been detected from these projects using the Arcan [19] tool, while the analysis of code smells has been carried out with SonarQube using the "Antipatterns-CodeSmell" plugin. The results of this step are the list of architectural smells and code smells present in each project analyzed. The raw data are available in the replication package [152].

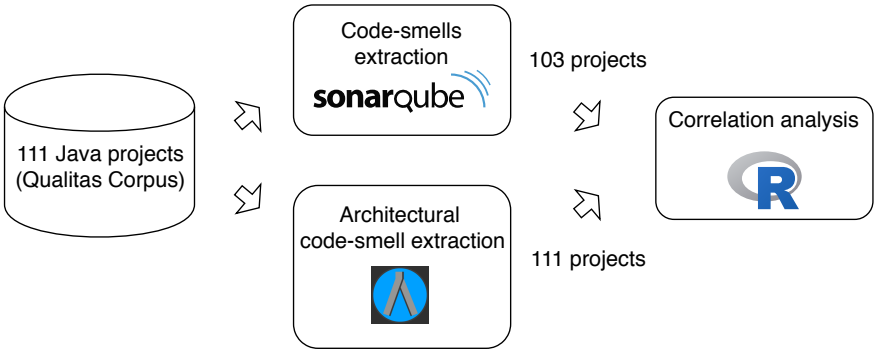


Figure 6.5: Data Collection Process

Code smell detection data

The SonarQube "Antipatterns-CodeSmell" plugin is a code smell detection tool that integrates DECOR (Defect dEtection for CORrection) [116] into SonarQube, detecting the 19 code smells reported in Section 6.2.1.1. DECOR can be applied on any object-oriented language, however, the SonarQube Plugin is only configured to detect code smells in Java. Moreover, SonarQube also calculates several other static code metrics such as the number of lines of code, cyclomatic complexity, but also reports code violations.

It is important to notice that in SonarQube (up to the current 6.5 version) the term ”Code Smells” is used to report coding style violations (also known as Issues in SonarQube) such as brackets closed on the wrong line, or redundant throws declarations. To avoid misunderstandings with coding style violations, the SonarQube "Antipatterns-CodeSmell" plugin tags all the code smells of Section 6.2.1.1 as "Antipatterns/CodeSmells". As for the detection accuracy, we rely on DECOR as detection tool since it ensures 100% recall for the detection of code smells [116]. Moreover, since the the definition of code smells is based on several metrics and thresholds, we rely on the standard metrics proposed by Moha et al [116] so as to ensure a precision average of 80%.

The code smells detection on the Qualitas Corpus dataset has been carried out on a Linux virtual machine with 4 cores and 16GB of RAM. The first 103 projects have been analyzed in 35 days while, because time constraint, we skipped the analysis of the remaining 9 projects such as Eclipse and JBoss, that would have taken more than three months. The reason of this dramatic increase of analysis time is due to the project structure. These nine projects are composed of several sub-projects with sizes similar to the other 103 project already analyzed. Therefore, in this work we only consider the results of the 103 projects listed in Appendix A.1.

6.2.3.1 *Architectural smells detection data*

All the architectural smells have bee detected wihth Arcan. The detection techniques of the AS and the validation of the tool results are described in previous Chapters 4 and 5. In these works, the results of the tool have been

validated on 10 open source systems and on two industrial projects obtaining a high precision value of 100% in the results. Moreover, the results of Arcan have been validated through a mixed method research. This last evaluation shows that the tool is precise – since it detects 100% of the architecture smells effectively present in the analyzed source code – and is reliable – since it detects the *correct* architecture smells in over 84% of the cases [135].

The architectural smells detection has been performed by using a Windows machine with 4 cores and 24 GB of RAM. The entire Qualitas Corpus dataset has been analyzed using Arcan in less than 24 hours.

6.2.4 *Data Analysis*

In this section, we describe the procedure we followed to analyze the collected data in order to answer our research questions.

We analyzed the classes infected both by an architectural smell and one or more code smells at class and package levels.

Architectural smells involve more than one Java class, while the 19 code smells considered in this work, involve only one class. Therefore, for each architectural smell, we could have one or more code smell infecting the same set of classes. In the analysis we calculated correlations only between code smells infecting the same classes (and packages) infected also by architectural smells.

As example, *classes A*, *B* and *C* can be infected by a Cyclic Dependency while *classes A* and *C* can be infected by God Class and *class D* infected by Speculative Generality. In this case, we calculate correlation only for the architectural smell Cyclic Dependency and the code smell God Class, since they are affecting the same set of classes, while we do not consider the code smell Speculative Generality, since it infects a class that is not also infected by Cyclic Dependency.

Before answering our RQs, we analyzed the distribution of the code smells and architectural smells in our data-set. We performed a descriptive analysis of the collected data analyzing the number of code smells and architectural smells per project and per package.

We analyzed the frequency of occurrences of the code smells and architectural smells in the 103 projects, considering:

- **(CS+AS)**: Classes infected by Code Smells **AND** Architectural Smells;
- **(CS)**: Classes infected **only by** Code Smells;
- **(AS)**: Classes infected **only by** Architectural Smells;
- **(HC)**: Healthy Classes - classes neither infected by code smells nor by architectural smells.

Projects without code smells or architectural smells were not considered for the analysis.

In order to answer our research questions, we applied the following analysis techniques:

- We tested the data for the normality by means of the Shapiro-Wilk test.
- If the data was normally distributed, we calculated Pearson correlation coefficient between code smells (or category of smells) and architectural smells (or Multiple Architectural Smell).
- If the collected data was not normally distributed, we calculated Kendall rank correlation coefficient between code smells (or category of smells) and architectural smells (or Multiple Architectural Smell).

We only considered correlation coefficient values with p-value smaller than 0.05, as customary done in empirical software engineering.

6.2.5 *Results*

In this section, we first describe the data analyzed and then we answer our research questions by reporting the results of the analysis described in Section 6.2.4.

All the projects contain classes infected both by architectural smells and code smells.

Considering the presence of code smells in the 103 projects, only 15 of the 19 code smells detectable by the SonarQube plugin were found. The 103 projects were not infected by Blob Class, Functional Decomposition, Base class Knows Derived and Tradition Breaker. This impacted also the categories of code smells containing the code smells not found in the projects since since two categories (Change Preventers and Object Orientation Avoiders) were based on two code smells not detected in the 111 projects. Therefore, the results of the analysis only considered the remaining four categories of code smells.

As for the architectural smells, Arcan detected them in 102 projects over 103 (Jasml contains no architectural smells). Therefore, we consider the set of 102 projects for the analysis.

Table 6.6 shows the number of projects infected by code smells, categories of code smells and architectural smells (Column #Inf.prj.), while the remaining columns report descriptive statistics. Please, note that for reason of completeness we also report data for the code smells categories based only on one code smell. However, these categories will not be considered in the next analysis so as to avoid duplications of the results.

Figure 6.6 shows the number of classes infected by code smells and architectural smells (CS+AS), classes infected only by code smells (CS), classes infected only by architectural amells (AS) and healthy classes, classes without smells (HC). Moreover, Figure 6.7 shows the distribution of the same data per package.

As reported in Table A.1 (Appendix A.1), Complex Class, Long Method and Long Parameter List were the most commonly detected code smells in the projects.

As for the architectural smells, all the projects were infected by at least two architectural smells. The analysis reported that 101 projects were infected by

Cyclic Dependency, 100 were infected by Hub-Like Dependency, 95 were infected by Unstable Dependency and 102 were infected by a Multiple Architectural Smell.

Table A.1 (Appendix A.1), reports the details on the number of code smells and architectural smells detected in each project.

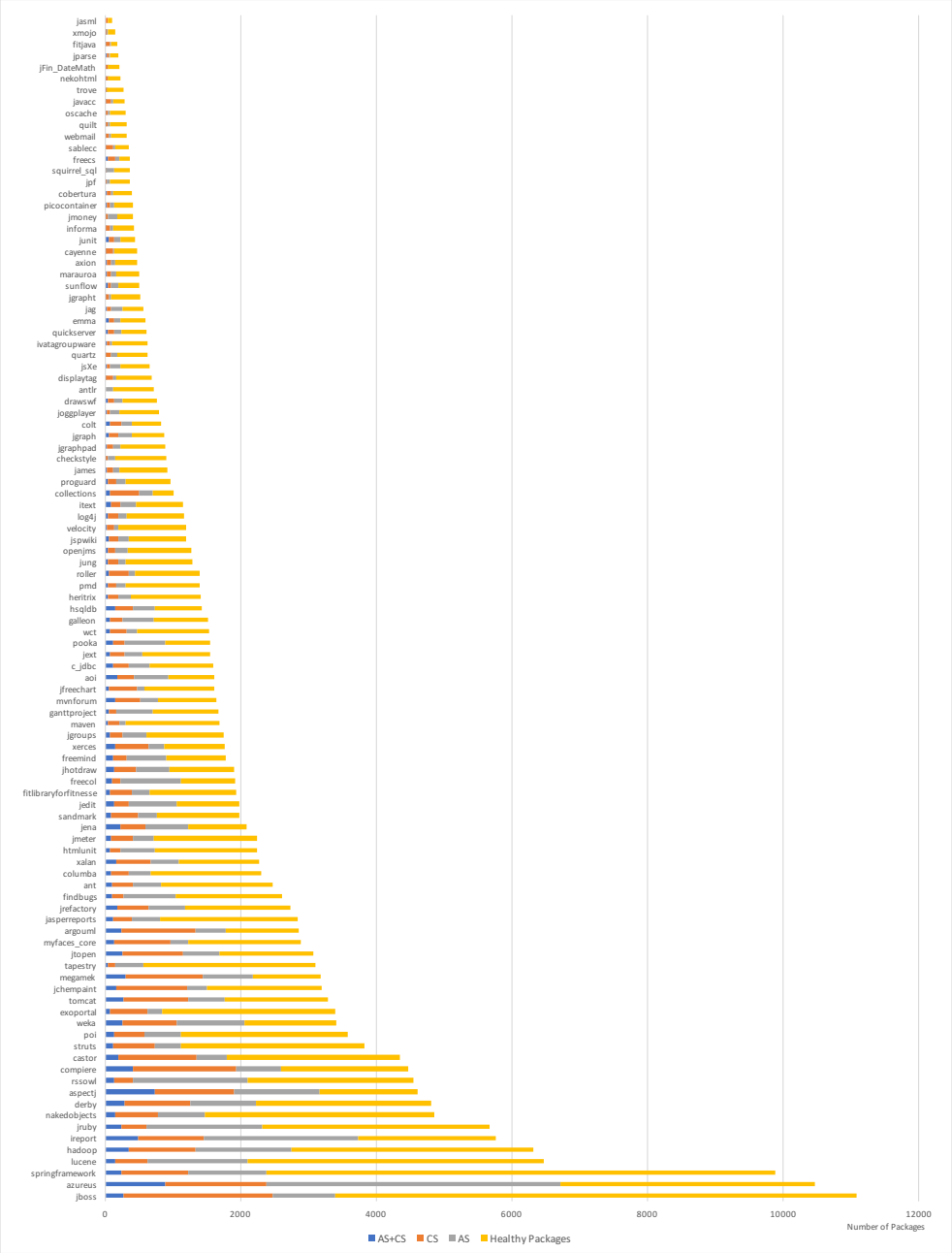


Figure 6.6: Number of packages infected by code smells or architectural smells

In Table 6.7, Table 6.8, Table 6.9 and Table 6.10 we report the results obtained analyzing the pairs AS-CS, while in Table 6.11 we present the results for the pairs AS-code smells category. These Tables report the number of infected projects for each pair (column *"#Inf. Prj."*), the number of infected projects where the results are statistical significant and their percentage up

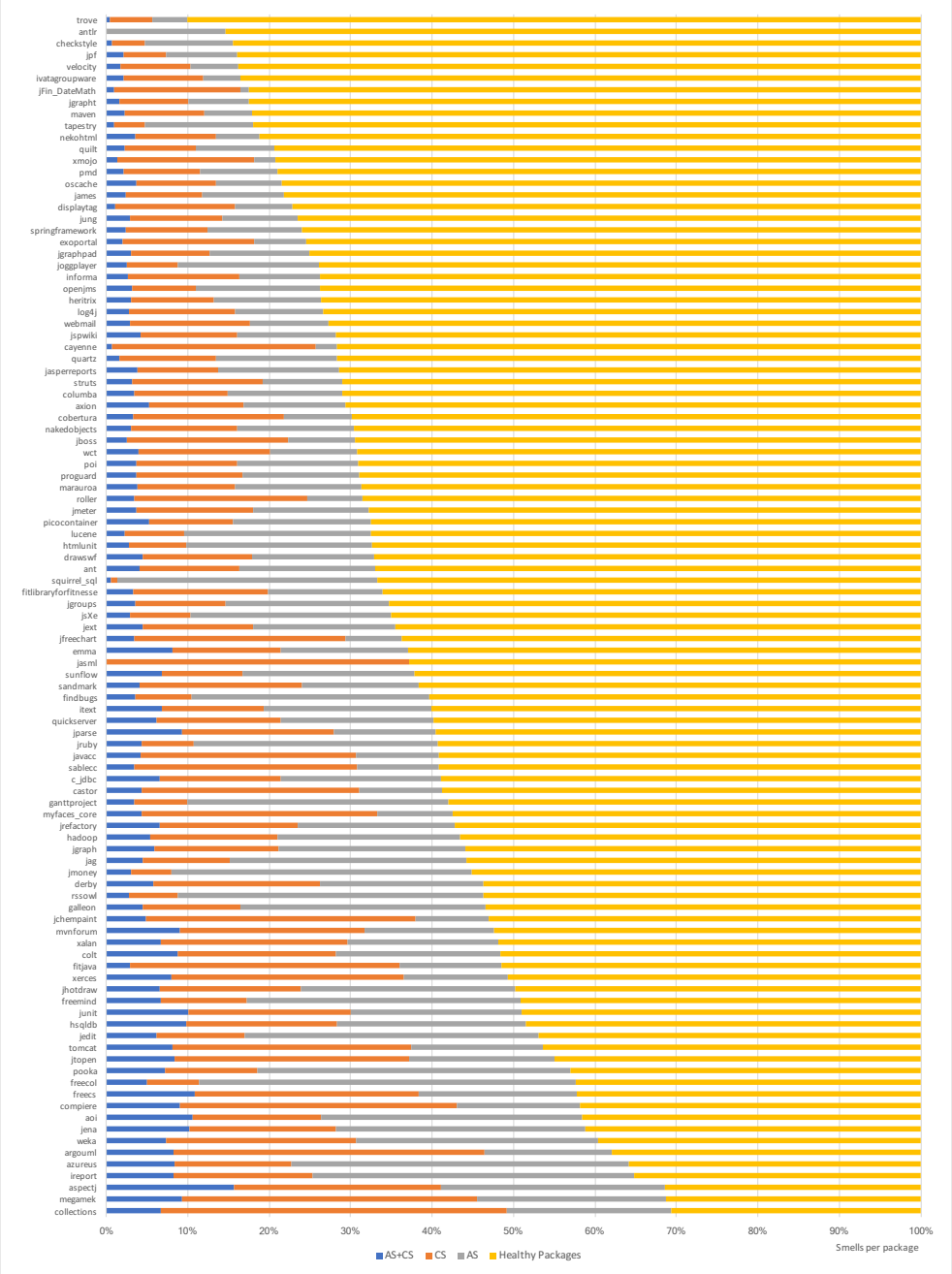


Figure 6.7: Number of code smells and architectural smells per package

to the total number of infected projects (column*"#Prj.(p<0.05)"*). Moreover, we also list the projects that reported a Kendall correlation higher than 0.5 (column*"#Prj.(tau<0.5)"*).

As example (Table 6.9), the pair composed by the architectural smell *Unstable Dependency* (UD) and the code smell *Base Class Should be Abstract* (BCSA) is detected in 54 projects (column *"#Inf.prj"*) and 30 of them (55% of projects) provide a significant statistical correlation with p-value <0.05 (column *"#Prj.(p-value<0.05"*). However, only 2 projects have a correlation higher than 0.5 (Column *"#Prj. (tau >0.5)"*) while the remaining ones (28 projects), not

listed in the Table, reports a statistical significant result with a low correlation (tau <0.5). Column "*Project*" points out the two projects with correlation higher than 0.5.

Table 6.6: Projects Infected by Code Smells, category of Code Smells or Architectural Smells

Name	#Inf.prj.	per Project AVG	Max	Min	StD
Code Smells					
Complex Class	103	147.90	914	1	163.23
Duplicated Code	103	237.28	1830	0	357.67
Long Method	102	178.88	1,251	0	197.58
Long Parameter List	100	94.09	1,197	0	157.51
Antisingleton	92	31.96	7.34	0	81.86
Class Data should be Private	90	28.93	3.53	0	50.07
Lazy Class	86	26.96	210	0	43.65
Spaghetti Code	58	2.97	40	0	5.23
Baseclass Abstract	54	3.84	65	0	8.49
Refused Parent Bequest	42	6.38	139	0	19.33
Speculative Generality	36	2.68	35	0	5
Many Field Attr. not Complex	32	0.76	20	0	2.23
Swiss Army Knife	11	1.39	76	0	8.22
Message Chains	8	1.27	62	0	7.19
Large Class	5	0.07	2	0	0.32
Baseclass Knows Derived	0	-	-	-	-
Blob Class	0	-	-	-	-
Functional Decomposition	0	-	-	-	-
Tradition Breaker	0	-	-	-	-
Category of Code Smells					
The Bloater	103	421.70	3,364	1	496.13
The Dispensables	102	264.24	1,849	0	379.22
The Obj.-Orientation Abusers	92	31.96	734	0	81.86
The Change Preventers	58	2.97	40	0	5.23
The Encapsulators	8	1.27	62	0	7.19
The Obj.-Orientation Avoiders	0	-	-	-	-
Architectural Smells					
Multiple Architectural Smell	102	6,148.02	162,531	0	22,176.7
Cyclic Dependency	101	6,122.24	162,357	0	22,162.1
Hub-Like Dependency	100	21.35	168	0	25.43
Unstable Dependency	95	4.43	15	0	3.16

Table 6.7: Projects Infected by Cyclic Dependency architectural smell (CD) and code smells (RQ1)

AS	CS	#Inf.prj	Prj.(p-value<0.05)		Prj.(tau>0.5)	
			#	%	#	prj. name
CD	ASG	92	70	76	1	xmojo
	BCSA	54	45	83	0	-
	CC	102	92	90	1	freecs
	DC	102	87	85	0	-
	DsP	90	60	67	0	-
	LC	5	1	20	0	-
	LM	102	87	85	0	-
	LPL	100	80	80	1	jparse
	LzC	86	28	32	0	-
	MFnC	32	10	31	0	-
	MC	8	7	87	0	-
	RPB	42	26	62	0	-
	SC	58	40	69	1	xmojo
	SG	36	22	61	0	-
	SAK	11	6	54	0	-

6.2.6 *Discussion*

In this Section, we answer our Research Questions (RQs) based on the results obtained and described in Section 6.2.5 and we draw the main lessons learned of this work.

RQ1:Is the presence of an architectural smell independent from the presence of code smells?

The results for RQ1 are presented in Table 6.7, Table 6.8 and Table 6.9. We analyzed 45 combinations (pairs CS-AS composed by 15 code smells and 3 architectural smells) for each of the 102 projects for a total of 4590 analysis.

We decided to not consider the combination (CD-LC, CD-MC, CD-SAK), (HD-LC, HD-MC and HD-SAK) and (UD-LC, UD-MC and UD-SAK) for the low number of infected projects (less than 12). We found statistical significant results (p-value<0.05) for all the other combinations.

However, only 14 combinations in 9 projects show a correlation higher than 0.5. Moreover, the same combination of code smell - architectural smell is found in a maximum of two projects.

For the other 40 combinations we found low correlations (tau<0.5) in a good number of projects (at least 32).

The results confirm our Hypothesis 1, since, based on the results of the 102 analyzed projects, we cannot identify dependencies between architectural smells and code smells.

Table 6.8: Projects Infected by Hub-like Dependency architectural smell (HD) and code smells (RQ1)

AS	CS	#Inf.prj	Prj.(p-value<0.05)		Prj.(tau>0.5)	
			#	%	#	prj. name
HD	ASG	92	80	89	1	jmoney
	BCSA	54	50	92	2	checkstyle, jparse
	CC	102	95	90	0	-
	DC	102	91	89	0	-
	DSP	90	80	89	2	checkstyle, jparse
	LC	5	5		0	-
	LM	102	94		0	-
	LPL	100	93		0	-
	LzC	86	78		0	-
	MfNC	32	26	81	1	checkstyle
	MC	8	7		0	-
	RBP	42	37	88	1	checkstyle
	SC	58	50	69	1	xmojo
	SG	36	31		0	-
	SAK	11	9		0	-

RQ1.1:Is the presence of a Multiple Architectural Smell independent from the presence of code smells?

The results for RQ1.1 are presented in Table 6.10. We analyzed 15 combinations (pairs CS-MAS composed by 15 code smells and a Multiple Architectural Smell) for each of the 102 projects (1530 analysis).

We decided to not consider the combination (MAS-LC), (MAS-MC) and (MAS-SAK) for the low number of infected projects (less than 12). We found statistical significant results (p-value<0.05) for the remaining 13 combinations. However, only 2 combination in 2 different projects show a correlation higher than 0.5. For the other 11 combinations we found low correlations (tau<0.5) in a good number of projects (at least 32).

The results confirm our Hypothesis 2, since the presence of a Multiple Architectural Smell does not depend from the presence of code smells in the 102 analyzed projects.

RQ2:Is the presence of an architectural smell independent from the presence of a category *of code smells?*

In order to answer this RQ, we considered the three categories of code smells reported in Section 6.2.1.2: *The Bloaters* (Bloat.), *The Dispensables* (Disp), and *The Object Orientation Abusers* (OOA). In this case, we considered all the code smells belonging to the same category as a single code smell.

The results obtained for RQ2 are shown in Table 6.11. We analyzed 9 combinations (pairs AS-CS composed by 3 category of code smells and 3 architectural smells) for each of the 102 projects (918 analysis).

Table 6.9: Projects Infected by Unstable Dependency architectural smell (UD) and code smells (RQ1)

AS	CS	#Inf.prj	Prj.(p-value<0.05)		Prj.(tau>0.5)	
			#	%	#	prj. name
UD	ASG	92	60	65	1	nekohtml
	BCSA	54	30	55	2	log4j, picocontainer
	CC	102	92	90	0	-
	DC	102	84	82	0	-
	DsP	90	63	70	0	-
	LC	5	4	80	0	-
	LM	102	82	80	0	-
	LPL	100	68	68	0	-
	LzC	86	36	42	0	-
	MFnC	32	9	28	0	-
	MC	8	5	62	0	-
	RBP	42	23	55	0	-
	SC	58	30	52	1	oscache
	SG	36	17		2	log4j, picocontainer
	SAK	11	8	72	0	-

We found statistical significant results (p-value<0.05) for all the combinations. However, only 2 combination with the same category of code smells (OOA) show a correlation higher than 0.5 in 2 different projects Table 6.11. For the other 7 combinations we found low correlations (tau<0.5) in a huge number of projects (at least 92). Results are similar to the one reported for the analysis of the non-grouped code smells (Table 6.7, Table 6.8 and Table 6.9).

We can accept our Hypothesis 3, since the presence of an architectural smell does not depend from the presence of a *category* of code smells. Even if three projects are infected by the same category of code smells, we cannot consider the results since the sample is too small.

RQ2.1:Is the presence of a Multiple Architectural Smell independent from the presence of a category *of code smells?*

In order to answer this RQ, we consider the same category of code smells adopted in RQ2.

The results obtained for RQ2.1 are shown in Table 6.11. We analyzed 3 combinations (pairs category of CS-AS composed by 3 category of code smells and 1 Multiple Architectural Smell) for each of the 102 projects (306 analysis). We found statistical significant results (p-value<0.05) for all the combinations. However, only 1 combination shows a correlation higher than 0.5 only in 1 projects. For the other 3 combinations we found low correlations (tau<0.5) in a huge number of projects (at least 98).

Table 6.10: Projects Infected by Multiple Architectural Smell (MAS) and code smells (RQ1.1)

AS	**CS**	**#Inf.prj**	**Prj.(p-value<0.05)**		**Prj.(tau>0.5)**	
			#	%	#	prj. name
MAS	ASG	92	66	72	0	-
	BCSA	54	32	60	0	-
	CC	102	90	88	0	-
	DC	102	64	63	0	-
	DsP	90	56	62	0	-
	LC	5	0	0	0	-
	LM	102	83	81	0	-
	LPL	100	80	80	1	jparse
	LzC	86	33	38	0	-
	MFnC	32	7	22	0	-
	MC	8	7	87	0	
	RBP	42	21	50	0	-
	SC	58	36	62	1	xmojo
	SG	36	21	58	0	-
	SAK	11	7	63	0	-

We can accepted the Hypothesis 4. The presence of a Multiple Architectural Smell does not depend from the presence of a *category* of code smells.

6.2.7 *Lessons Learned*

Lesson Learned 1: An architectural smell or multiple architectural smells do not depend from code smells. As we can see from the analysis, in all cases, for all the couples AS-CS, except MAS-LC (Multiple Architectural Smell-Large Class), we found statistic significant results in all the projects. However, some code smells are more frequently infecting projects and therefore, the results of the analysis are more reliable for them. Considering all the analysis (4590 for RQ1 and 1530 for RQ1.1) 58.8% of them provided statistical significant results and only 0.03% of them have shown a correlation higher than 0.5.

Lesson Learned 2: An architectural smell or multiple architectural smells do not depend from categories of code smells. Also in this case, 78% of them provided statistical significant results (918 for RQ2, and 306 for RQ 2.1) and only 0.3% of them have shown a correlation higher than 0.5.

Therefore, the main lesson learned from the analysis of the RQs is that architectural smells do not depend on code smells and therefore, the refactoring of code smells does not decrease the chances to remove architectural smells. Moreover, also the removal of an architectural smell does not imply the removal or the reduction of code smells.

Table 6.11: Projects Infected by architectural smells (RQ2) or Multiple Architectural Smells (RQ2.1) and by categories of code smells

AS	CS cat.	#Inf.prj	Prj.(p-value<0.05)		Prj.(tau>0.5)	
			#	%	#	prj. name
CD	Bloat.	103	91	88	0	-
	Disp.	102	74	72	0	-
	OOA	98	73	75	0	-
HD	Bloat.	103	95	92	0	-
	Disp.	102	90	88	0	-
	OOA	92	87	94	1	jmoney
UD	Bloat.	102	87	85	0	-
	Disp.	102	68	67	0	-
	OAA	98	69	70	1	nekohtml
MAS	Bloat.	102	88	86	1	jparse
	Disp.	102	68	67	0	-
	OOA	98	66	67	0	-

6.2.8 *Threads to Validity*

In this Section, we introduce the threats to validity, following the structure suggested by Yin [164] reporting construct validity, internal validity, external validity, and reliability. Moreover, we also debate the different tactics adopted to mitigate them.

Construct Validity concerns the identification of the measures adopted for the concepts studied in this work. For this threat, the main issue is related to the detection accuracy of the adopted tools. For this purpose, we relied on existing detection tools already adopted in research works. As for the code smells detection, we relied on DECOR rules. Please, consider that our SonarQube "Antipatterns-CodeSmell" plugin adopts the exact rules defined by Moha et al. [116]. We are aware that the results could be influenced by the presence of false negatives and positives. For this reason, Moha et al. reported for DECOR a precision higher than 60% and a recall of 100% on a selected set of projects. Moreover, in our previous work [153], two authors have independently manually validated a subset of smell instances reporting a mean precision of 78%. The results of the validation analyzed in [153] are also available in its replication package [152]).

The evaluation of Arcan detection performances in two industrial case studies done through the feedbacks of the developers is described in [19], where the authors report a precision of 100%, since Arcan found only correct instances of architectural smells. The developers reported 5 more architectural smells which were false negatives related to 180 external components out of the scope of the analysis of the tool.

Moreover, the correctness of Arcan detection with respect to its ability to detect the *right* architecture smell in the right class/package, with respect

to a given, independently pre-coded dataset and the completeness of Arcan results with respect to its ability to detect *all* instances of architecture smells with respect to the same dataset is reported in [135].

Based on the previous assumptions, the presence of false positives and false negatives is mitigated by the large sample of analyzed projects and by the very high precision and recall values of the two detection tools results.

Threats to *Internal Validity* concern factors that could have influenced the results obtained.

In order to reduce this threat due to the context, we analyzed a set of 102 well-known Java projects included in the Qualitas Corpus dataset. This dataset includes projects from different domains, of different sizes and with different architectures.

Threats to *External Validity* concern the generalization of the results obtained. We cannot claim that our results fully represent any Java project. In order to mitigate this issue, we considered a large set of projects with different characteristics.

Threats to *Reliability* refer to the correctness of the conclusion reached in the study. We applied non-parametric tests and rank-based correlation methods since software metrics do not have often normal distributions. We used a standard R packages for performing all statistical analyses since it allows simple replications of them and gives a good confidence on the quality of the results.

6.2.9 *Conclusions*

In this study, we conducted a large-scale empirical study investigating the relations between code smells and architectural smells and if code smells affect the presence of architectural smells and vice versa. We detected code smells and architectural smells on 102 Java projects of the Qualitas Corpus dataset [156] by means of two smells detection tools, the SonarQube "Antipatterns-CodeSmells plugin" for code smells and Arcan for architectural smells.

We found empirical evidence on the independence between code smells and architectural smells and therefore, we can suppose that the presence of code smells does not imply the presence of architectural smells. Hence, for a developers/maintainer the refactoring of code smells does not led to the removal of architectural smells.

On this study a paper in collaboration with the Free University of Bozen-Bolzano (Italy) has been submitted to IST Elsevier Journal [20].

7 PROPOSAL OF A NEW ARCHITECTURAL DEBT INDEX

Many works have been proposed in the literature on managing technical debt [89, 118, 157] and different forms of technical debt can be considered at different levels, e.g., code, architecture, test, social, documentation and technological.

As outlined and evaluated by Ernst et al. [47] architectural issues are the greatest source of technical debt. Hence, it is important to understand how to manage architectural concerns to avoid technical debt accumulation. In this chapter, I focus my attention on a possible evaluation of architectural debt [118].

7.1 DISCUSSION ON THE MAIN TDI FEATURES

In this section, the most relevant features and differences found in the indexes provided by the tools (described in Section 2.5) are discussed, with the aim to understand what exactly each index takes into account, what the value of the index represents, and its completeness w.r.t. the information that can be exploited to estimate Technical Debt. With these aims, the following questions were defined:

Q1 How are the quality indexes the tools provide exactly computed? Which features do they take into account?

Q2 Which index does take more into account architectural issues and in which way?

Q3 Which are the features not provided or taken into account by the indexes?

Tab. 7.1 (to answer Q1, Q2, Q3) shows the different categories of input information used by the tools to compute their indexes. Tools may also support the extraction of some information that is not used to compute a TDI: only the information used in the index computation were considered. For example, past versions of SQ were able to detect some architectural issues, but not used in the index computation, while it uses the wider range of code level information. All the tools detect an architectural smell, Cyclic Dependency. Only inFusion detects two more architectural smellsi, but the tool is not available anymore.

In Tab. 7.2 (respect to Q1 and Q3), the information provided by the different TDIs are characterized, regarding both what the measures address and how they represent it (output information). The table shows that indexes do not always provide estimation of both the costs of correcting the system (Resolution cost, i.e., the TD principal) and of keeping it unchanged (Keeping

Table 7.1: Input information of Technical Debt Indexes used by tools

Information category	CAST	IF	SG	SQ	S101
Architectural Smells, e.g., [52, 93]	yes	yes	yes	no	yes
Cyclic Dependency smell	yes	yes	yes	no	yes
Other architectural smells	no	yes (2)	no	no	no
Code Smells [48, 80]	no	yes	no	yes	no
Architecture/Design Metrics, e.g., [109]	yes	no	no	no	yes
Code Metrics, e.g., [38, 80]	yes	no	no	yes	yes
Architectural Violations$^{\alpha}$	yes	no	yes	no	no
Coding Rule Violations$^{\beta}$	yes	no	no	yes	no

α: deviations from a reference architecture, i.e., unallowed dependencies
β: detected bad coding practices or excessive values of single metrics (some tools, e.g., SQ, internally refer to the latter as "smells")

Table 7.2: Output of Technical Debt Indexes provided by tools

	CAST	inFusion	Sonargraph		SonarQube			Structure101
TDI name →	TDP	QDI	SDI	SDC	TD	TDR	SR	XS
Resolution cost	yes	no	no	yes	yes	yes	yes	no
Keeping cost	no	yes	yes	no	no	no	no	yes
Unity Measure	US$	-	-	US$	Time	-	Rank	LOC

cost, i.e., the TD interest). These two aspects of TD are highly relevant during estimation, and this should be seen as an important improvement opportunity for tool vendors. A particular issue can be found in SonarQube, which implements the SQALE method for TD estimation, where the costs of keeping an issue are captured by the SQALE Business Impact Index (SBII). This index is not available in the free version of SonarQube, while it may exist in the commercial one. Anyway, the SonarQube classification of issues using the Info–Blocker range can be seen as a non-quantitative suggestion of the non-remediation costs.

A similar choice has been done in CAST, by using a three-value severity (Low, Medium, High) to classify the detected problems. In both cases, the severity of each issue/problem has been defined by the authors of the tool, and can be customized by advanced users.

In Sonargraph, the SDI is computed as a score, proportional to the set of issues considered in the index computation. This makes the Structural Debt Index (SDI) proportional to the estimated quality of the system, i.e., an estimation of Keeping costs. Structural Debt Cost (SDC) is the money cost of fixing the system, i.e., a Resolution cost, but its value is computed linearly from the SDI, making the distinction between them not very clear from this point of view.

Finally, different indexes provide different kinds of estimation. SonarQube's TD computes its value directly in terms of time needed to fix the

reported issues. Then, it provides derived indexes with the aim of making comparisons among different projects easier, since the absolute TD value will be higher in larger projects, but its "density" should be kept under control. All the other indexes start instead from computing a score based on the count or size/severity of the issues detected in the analyzed project. In CAST and Sonargraph there is a mechanism that weights the effort associated to the computed score to express it in terms of time and costs. inFusion and Structure101 do not provide this mechanism. inFusion expresses its index using an abstract number. A peculiarity of Structure101 is to express its XS metric in terms of LOC affected by some detected issues. This choice allows relating the XS value to the size of the system, and to have an idea of how much the detected issues are widespread in the system.

All the Indexes share a common rationale, i.e., they start from the evaluation of specific quality indicators, e.g., architectural violations, code smells, and weight their relative severity, aggregating the outcome to provide an overall evaluation of the entire project or of its different parts. Anyway (always w.r.t Q3), the association of resolution times to issues is arbitrary. Tools allow customizing the effort associated to single issues (SQ) or to their scores (SG, CAST), but these values are arbitrary and there is no established guideline on how to set them. This is probably the reason why the other tools, other than SQ, did not associate effort measures to their indexes, and prefer to keep them as indexes to be used mainly when comparing the quality of a system in different revisions.

Below, I propose the answers to the three questions:

Q1 - The tool takes mainly into account metrics, smells, coding rules violations and architecture violations, as shown in Tab. 7.1. The estimations they provide are different in terms of unity of measure (e.g., time, cost, abstract numbers) and TD target (remediation costs and keeping costs), as shown in Tab. 7.2.

Q2 - Given the discussion on the indexes provided in Sec. 7.1, it is possible to observe that: *1)* Sonargraph, Structure101 and CAST use the largest share of architectural information in their indexes but they are able to detect only one AS; *2)* SonarQube does not use architectural information when computing its index.

Q3 - First, no tool uses all the information that can be exploited, at both code and architectural level, and no tool provides both Keeping and Resolution costs. Hence, tools could try to fill the gaps in their estimation models by re-using some knowledge exploited by other tools. Another observation is that tools are conservative in their estimation features, by relying only on static analysis and without exploiting historical information about the analyzed projects. In particular, most tools allow showing the history of their analyses on different revisions of the same project, but none of them uses the underlying changes in the software itself to spot issues relevant to Technical Debt estimation. For example, the detection of evolutionary coupling can be used to discover unseen dependencies, as done through the ICPD smell detected by Arcan.

In most cases, the available Indexes are not directly useful when evaluating a single project. The provided measures cannot be interpreted with the aim to understand the overall quality of the analyzed project on a global scale. As a consequence, these Indexes should in particular be useful on a relative scale, in the case a single team evaluates an entire portfolio of applications. In this case, the Index can be used, e.g., to rank new projects w.r.t. the old/existing ones.

Finally, as already observed different architectural smells are not taken into account.

7.2 A NEW ARCHITECTURAL DEBT INDEX

As outlined before, the TDIs computed by different tools are often more focused on code issues than architectural ones. Many architectural problems such as architectural smells are not considered; the same happens with the relations (structural or statistical) existing among code and architectural problems. Moreover, different architectural smells can be identified only by analyzing the development history of a project, and Technical Debt (TD) indexes do not take into account this kind of information too.

The aim of the following sections is to answer the research questions described below:

RQ1 How should a new index be formulated to better represent architectural debt?

RQ2 How can we estimate the severity of an architectural smell?

RQ3 How does the evolution of the new index behave though the analysis of projects history? and how does it behave with respect to another index?

The answer to research question *RQ1* gives and motivates the definition of a new index, called *Architectural Debt Index (ADI)* based only on the detection of different architectural smells (AS) through the tool Arcan, described in Chapter 4. The computation of the ADI is implemented in Arcan through the detection of all the architectural smells of a given project.

The answer to research question *RQ2* is given through a severity estimation based on the the architectural smells detected by Arcan. The severity is defined according to the detection algorithm of each architectural smells.

The answer to research question *RQ3* reports the evaluation of ADI on a large dataset of open source projects. Moreover, the evaluation of the ADI on more than 100 versions of 10 projects is analyzed and it is compared with the index implemented in another tool: SonarQube.

7.3 WHICH INDEX SHOULD BE DEFINED?

The aim of the ADI is to provide through a unique value an overall evaluation of the quality of a project. It can be used to compare different projects,

evaluate their quality during their evolution and to identify weak spots in a software system. The definition of a new index aims to give some hints on how to improve projects through the evaluation and analysis of different factors:

- *which are the oldest AS which persist in the project? and the new ones?* through the evaluation of the history of a project it is possible to get this data and perform empirical analysis to check if:
  - the persistent smells are also the more critical ones;
  - the new ones are the easiest to remove or not;
- *which are the most critical AS which should be removed first?* Since architectural smells point to possible problems in real projects leading to a high evolution cost and a progressive architecture degradation (low quality), it is important to establish which ones must be removed first. Three priority levels can be considered for the smells to be removed:
  1. *high priority*: remove first the more critical smells (*cost-presence*) and easier to solve (*cost-solving*);
  2. *medium priority*: remove the less critical smells and easy to solve;
  3. *low/no priority*: remove the less critical smells or/and difficult to solve.

  I define *Cost-presence* (or keeping cost) as the cost of keeping the TD in the project. I'll propose to compute this cost through an evaluation of a Severity Score included in the computation of the ADI index. The *Cost-solving* (or remediation cost) is the cost (e.g., time, money, energy) to solve or remove the TD and can be defined by taking into account several aspects, such as the effort needed for the refactoring of an AS which can be computed in different ways. Regarding the AS detected through Arcan, we could consider:
  - the number of dependency relations to be cut in order to remove a *Cyclic Dependency* smell;
  - the number of classes to be moved from a package to another one, to remove *Unstable Dependency* smell;
  - the number of classes/packages that a class/package affected by *Hub Like Dependency* smell must be broken up into.
  - the number of dependency relations to be cut in order to remove a *Specification-Implementation Violation* smell;
  - the number of files to be moved to a package, to remove *Implicit Cross Package Dependency* smell;

  This list is composed by the architectural smells detected by Arcan, other features can be taken into account for other AS according to their features.

7.4 AN ARCHITECTURAL DEBT INDEX

This section is focused on the evaluation of the *cost-presence* of architectural smell through the definition of a new index based only on the detection of the AS currently identified through Arcan.

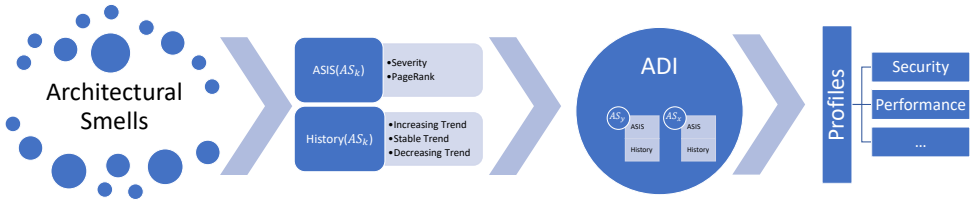


Figure 7.1: ADI workflow

The workflow of the ADI computation for a project is shown in Figure 7.1 and it is composed by 4 main steps: *1)* the detection of architectural smells, *2)* the architectural smell severity estimation and the evaluation of the AS according to their History, *3)* the computation of the ADI of the project and *4)* the ADI computation according to different quality index profiles.

The implementation of the last step belongs to a future development, but the ADI-profile proposal is described below (Section 7.4.5).

The index computation takes into account the *Number* of AS in a project, i.e., whether the number of AS grows, the Index increases (lower quality) for the project.

The computation considers also the evaluation of the AS, considering:

- the *Severity* of an AS: assuming that some instances of AS are more critical than others, the Index takes into account a Severity measure defined according to each architectural smell type;
- the *History* of AS: the presence of an AS in the history of a project can have a different impact on the index. For example if an AS involves an increasing number of classes and packages in its evolution, it can be considered more critical than other AS.

Moreover, different metrics are taken into account implicitly , i.e., the metrics used for the AS detection, in particular the Dependency metrics of Martin [109] (Instability, Fan In, Fan Out, Efferent and Afferent Coupling).

By considering these factors, the Architectural Debt Index (ADI) of a project P is defined as follows:

$$ADI(\mathrm{P}) = \sum_{k=1}^{n} \left(\frac{1}{W} (ASIS(AS_k) * w(AS_k)) * History(AS_k) \right) \tag{7.1}$$

where:

- n: number of AS instances in a project P;
- AS_k: k-instance of an architectural smell;

- W: the total number of dependencies involved in at least one AS for all the AS in the project;
- $ASIS(AS_k)$: the *Architectural Smell Impact Score* (defined below);
- $w(AS_k)$: the *Architectural Smell Weight*, i.e., the number of dependencies associated to the AS_k;
- $History(AS_k)$: the score associated to the trend evolution of the AS_k (defined below);

The dependencies are the number of unique vertices (classes or packages) of the subgraph directly affected by an architectural smell. In sections 7.4.1 and 7.4.3, the *ASIS* and the *History* functions are described respectively.

7.4.1 *ASIS*

The *Architectural Smell Impact Score*, *ASIS*, is based on both the estimation of the severity of an AS and of the importance subsystem where the AS is found. It is defined as the product of the *SeverityScore* associated to the AS_k smell and the *PageRank* value of the AS_k, which estimates the importance of the project subsystem affected by the AS_k smell (defined below).

It is not possible to define a general formula for the Severity Score and the PageRank computation suitable for all types of Architectural Smells. In addition, since we are combining the two values linearly, we need to mitigate potential non-linearities in their distribution, avoiding masking effects due to extremely large or small values.

Hence, we decided to compute the quantiles of the two values on a large dataset of projects, and use them to assign values to some thresholds we define, such as: low, medium low, medium, medium high and high.

The *SeverityScore* and *PageRank* values used to compute *ASIS* are the quantiles of the effective values computed on a selected dataset of 109 projects described in Section 7.4.1. The *SeverityScore* and *PageRank*, defined below, will both assume values in range $[0, \infty)$ mapped to integer values in the range $[0, 1]$ (low to high respectively), through the $quantile(x)$ function which is the quantile associated to x in the reference dataset.

Hence, the *ASIS* is defined as follows:

$$ASIS(AS_k) = SeverityScore(AS_k) * PageRank(AS_k) \tag{7.2}$$

.

Since both $SeverityScore()$ and $PageRank()$ return values between 0 and 1, *ASIS* represents a *SeverityScore* weighted by the "importance" (*PageRank*) of the subsystem where it appears.

THE SEVERITY SCORE The $SeverityScore(AS_k)$ is a value defined for each instance of AS according to each type of AS.

The *SeverityScore* for the five AS detected by Arcan, Unstable Dependency(UD), Hub-like Dependency (HL), Cyclic Dependency (CD), Specification Implementation Violation (SIV), Multiple Architecture Smell(MAS), is defined as follows:

- If AS_k is a UD, $SeverityScore(AS_k)$ is defined as: $quantile(NumberOfUnstableDependencies)$
- If AS_k is a CD, $\forall$ e $\in$ edges(CD), $SeverityScore(AS_k)$ is defined as: $quantile(NumberOfelementsInCD * min(n_occ(e)))$ where the $n_occ(e)$ is the number of times the same type of edge among two vertices (e.g., class or package) occurs.of the edge e.
- If AS_k is a HL, $SeverityScore(AS_k)$ is defined as: $quantile(TotalNumberOfDependencies)$
- If AS_k is a SIV, $SeverityScore(AS_k)$ is defined as: $quantile(NumberoftheViolation)$
- If AS_k is a MAS, $SeverityScore(AS_k)$ is defined as: $quantile(Numberofarchitecturalsmellsdetected)$

THE PAGERANK $PageRank(AS_k)$ estimates whether the AS is located in an important part of the project, where importance is defined by the value of the PageRank algorithm executed on the dependency graph of the project (to evaluate if many parts(subsystems) depend on the part where the AS is involved). The PageRank, PR is modeled starting from the one implemented by Google [32], as explained below:

$$PR(v) = \frac{1-d}{N} + d\left(\sum_{k=1}^{n} \frac{PR(p_k)}{C(p_k)}\right) \tag{7.3}$$

where:

- the vertex v is a node of the dependency graph associated to a project;
- $PR(v)$ is the value of PageRank of the vertex v;
- N is the total number of AS in the project;
- P_k is a vertex with at least a link directed to v;
- n is the number of the p_k vertexes;
- $C(p_k)$ is the number of links of vertex p_k;
- d (damping factor) is a custom factor fixed at 0,85 (default value defined by Google [32]). It can be changed according to the PageRank value needed for every vertex and its minimum associated level of PageRank.

The PR value is computed only on vertices of the dependency graph of both class and package types. PageRank value PR is used for all the types of AS detected by Arcan, but the PageRank of an AS which involves multiple classes and packages is considered differently, e.g, CD smell involves two or more classes or packages. To compute the *PageRank* when an AS involves more than one vertex, it is necessary to aggregate the data; a method to aggregate multiple value could be take the maximum PR value of the group.

The PR of all the AS and the max of PR of AS involving multiple classes or packages is computed as follows:

$$PageRank(AS_k) = \begin{cases} quantile(max_{j=1}^{n} PR(v_j)) & \text{If } AS_k \text{ is an AS among classes or packages} \\ quantile(PR(v)) & \text{If } AS_k \text{ is an AS of a class or a package} \end{cases}$$

where v is the vertex (class or package) affected by AS_k, n is the number of classes v_j involved in an AS (among classes) or the number of packages v_j involved in an AS (among packages).

7.4.2 *The ASIS evaluation*

The distribution of the *SeverityScore* and *PageRank* is analyzed on a dataset of 109 projects of the Qualitas Corpus [156]: the quantile values of the dataset distribution are reported in Table 7.3, where the $SeverityScore(AS_k)$ and $PageRank(AS_k)$ assume values between 0 and 1. UD, HL and CD smells are considered in this analysis. However, ICPD and SIV smells are excluded since the ICPD is based on the history of a project and the SIV is out of scope of this primary case study. The Table 7.3 reports also the metrics used to compute the *SeverityScore* in according to Section 7.4.1: the number of unstable dependencies for UD; the number of total dependencies for HL; the number of vertices (i.e., number of classes of packages in according to the granularity level) and the number of involved cycle for CD.

Table 7.4 reports the values of the Architectural Debt Index (ADI) in the referenced dataset (without considering the factor related to the History of the projects) and its quantification as a score value in a range among 1 and 5 through the function $q(ADI(P))$:

$$q(ADI(P)) = \begin{cases} 1 & \text{If } 0.00 \leq quantile(ADI(P)) \leq 0.20 \\ 2 & \text{If } 0.20 < quantile(ADI(P)) \leq 0.40 \\ 3 & \text{If } 0.40 < quantile(ADI(P)) \leq 0.60 \\ 4 & \text{If } 0.60 < quantile(ADI(P)) \leq 0.80 \\ 5 & \text{If } 0.80 < quantile(ADI(P)) \leq 1 \end{cases} \tag{7.4}$$

where $quantile(ADI(P))$ is the quantile associated to the ADI computed for the project P of the reference dataset, e.g, given a project with $q(ADI(P))$ of 5, its ADI is worst than a project with a $q(ADI(P))$ of 2. Moreover, Table 7.4 shows every single factor involved in the computation of the ADI: $\Sigma ASIS$, $\Sigma ASIS * w$, W and the number of the AS found in the projects for the UD, CD and HL smells, both at package and class level.

From the data of Table 7.4 we can observe that projects with a high number of architectural smells have a higher $q(ADI(P))$, in fact Eclipse project has $q(ADI(P))$ of 5 and 20915 architectural smells; Cayenne has 8 architectural smells and has $q(ADI(P))$ of 1. Moreover, we can see that two projects with similar W (the number of dependency of the project) got different $q(ADI(P))$, such as Batik and Freecol got 3 and 5 respectively due to the different number of architectural smells and the higher $ASIS$. Batik and Findbugs have similar W and different $q(ADI(P))$ of 3 and 5 respectively, because the number of architectural smells are similar but the $\Sigma ASIS * w$ is bigger for Findbugs than Batik. Hence, Findbugs has bigger and worst architectural smells than Batik.

The JHotDraw and Tapestry projects have 389 and 398 architectural smells respectively, but they have $q(ADI(P))$ value of 1 and 2 respectively. Moreover, these values are lower than for projects with less architectural smells (e.g., JParse got the value 3 for $q(ADI(P))$), since the high number of architectural smells is mitigated mainly by the higher W.

Table 7.3: ADI's components Quantile and value associated

Quantile	Unstable Dep. Package			Hub-Like Dep. Class			Cyclic Dependency Class				Cyclic Dependency Package			
	PR	SS	NUD	*PR*	SS	TD	*PR*	SS	NOC	NOV	*PR*	SS	NOC	NOV
0.00	1.08	1	1	4.35	1	21	0.78	1	1	2	1.84	1	1	2
0.05	2.85	2	1	9.08	1	33	0.90	35	1	2	8.62	28	1	2
0.10	3.85	3	1	11.54	1	39	0.96	72	1	2	12.97	74	1	2
0.15	4.77	5	1	15.44	1	44	1.05	113	1	2	17.61	132	1	2
0.20	5.63	6	1	18.24	1	52	1.16	162	1	2	22.71	197	1	3
0.25	6.71	8	1	22.38	1	59	1.32	218	1	2	28.53	254	1	3
0.30	7.81	10	1	25.93	1	66	1.53	283	1	2	34.82	305	1	4
0.35	8.94	12	1	30.04	2	69	1.82	354	1	2	42.11	354	1	4
0.40	10.38	14	1	34.53	2	74	2.20	435	1	2	51.55	435	1	5
0.45	12.14	17	1	37.71	2	81	2.84	527	1	3	62.94	542	1	5
0.50	13.69	20	2	46.65	2	85	3.97	632	1	3	75.38	679	1	6
0.55	15.83	24	2	51.08	3	92	5.66	747	1	4	90.00	861	1	7
0.60	18.46	28	2	56.68	3	96	8.59	889	1	5	106.77	989	1	8
0.65	22.85	33	2	64.48	3	102	14.32	1069	1	7	126.19	1159	1	9
0.70	26.70	39	3	75.57	3	108	23.62	1325	1	10	150.30	1428	1	11
0.75	32.64	46	3	82.23	4	117	35.32	1664	1	14	185.52	1744	1	12
0.80	41.35	57	4	93.06	4	129	51.64	2052	1	18	256.20	2216	1	15
0.85	51.60	75	5	114.51	5	139	79.75	2498	1	24	380.28	3416	1	19
0.90	72.29	95	6	137.18	5	165	139.14	3360	1	33	553.30	4366	1	27
0.95	117.32	128	9	167.26	7	194	263.39	4920	2	54	836.93	5751	2	49
1.00	1766.90	207	42	428.95	11	893	418.22	7231	93	102	1599.88	7214	36	79

PR: PageRank, SS: Severity Score, NOC: Number of Cycle, NOV: Number of vertices, TD: Number of Total Dependency, NUD: number of unstable dependencies.

Table 7.4: Systems ADI computation and AS detection

	Index					Architectural Smells					
						UD	CD		HL		Total
System	$\Sigma ASIS$	$\Sigma ASIS * w$	W	ADI	$q(ADI)$	Pkg	Cl	Pkg	Cl	Pkg	AS
ant-1.8.2	263.1	2353.5	564	4.173	5	26	730	190	4	6	956
antlr-3.4	37.7	318.5	255	1.249	4	8	120	8	4	0	140
aoi-2.8.1	517.8	7506.6	656	11.443	5	7	1007	30	4	2	1050
argouml-0.34	85.5	620.9	489	1.270	4	25	305	114	0	3	447
aspectj-1.6.9	1655.3	35149.6	1476	23.814	5	52	2882	135	6	2	3077
axion-1.0-M2	6.2	20.7	67	0.309	2	5	30	23	0	1	59
azureus-4.7.0.2	3548.5	130507.8	4504	28.976	5	168	5497	1741	3	3	7412
batik-1.7	162.3	895.8	1141	0.785	3	34	617	92	8	2	753
castor-1.3.3	84.0	613.4	706	0.869	3	58	289	139	1	8	495
cayenne-3.0.1	0.4	1.8	77	0.024	1	1	4	1	1	1	8
checkstyle-5.6	20.8	79.2	185	0.428	2	3	77	6	2	2	90
c_jdbc-2.0.2	85.8	519.1	526	0.987	3	35	236	129	3	4	407
cobertura-1.9.4.1	1.2	3.0	111	0.027	1	4	14	5	4	2	29
collections-3.2.1	15.0	49.0	287	0.171	1	5	122	21	4	0	152
colt-1.2.0	56.7	306.6	241	1.272	4	9	173	42	4	2	230
columba-1.0	62.9	727.6	459	1.585	4	60	163	187	1	4	415
compiere-330	403.0	4992.6	926	5.392	5	19	922	111	4	5	1061
derby-10.9.1.0	434.0	5252.5	1063	4.941	5	51	1087	153	1	3	1295
displaytag-1.2	6.4	18.1	66	0.275	2	5	30	13	1	0	49
drawswf-1.2.9	25.4	119.2	160	0.745	3	14	87	15	2	0	118
drjava-20100913	761.5	8672.7	1818	4.770	5	14	1818	76	9	3	1920
eclipse_SDK-3.7.1	6266.9	92813.4	18058	5.140	5	608	18040	2258	3	6	20915
emma-2.0.5312	8.9	27.5	141	0.195	1	8	56	19	3	0	86
exoportal-v1.0.2	18.5	55.1	239	0.230	2	49	107	40	0	0	196
findbugs-1.3.9	411.0	5864.4	1106	5.302	5	13	909	105	8	2	1037
fitjava-1.1	3.0	9.4	69	0.136	1	0	12	0	2	0	14
fitlibrary-20110301	109.4	1080.5	351	3.078	4	38	181	172	1	5	397
freecol-0.10.3	1611.9	31660.1	1169	27.083	5	20	2479	124	11	4	2638
freecs-1.3.20100406	42.7	374.4	155	2.415	4	6	99	22	4	0	131
freemind-0.9.0	347.3	2696.9	846	3.188	4	16	837	82	4	5	944
galleon-2.3.0	125.5	686.9	672	1.022	3	6	529	26	5	3	569
ganttproject-2.1.1	141.0	788.1	700	1.126	3	20	484	99	2	3	608
geotools-9.2	562.9	6518.7	2446	2.665	4	206	1433	808	1	5	2453
hadoop-1.1.2	330.0	3326.0	1743	1.908	4	48	1120	298	4	5	1475
heritrix-1.14.4	49.7	354.5	369	0.961	3	16	137	108	3	3	267
hibernate-4.2.0	625.6	11485.3	1408	8.157	5	124	1038	636	0	3	1801
hsqldb-2.0.0	354.4	7324.4	427	17.153	5	11	651	45	7	2	716
htmlunit-2.8	410.0	7226.4	613	11.789	5	7	765	53	2	0	827
informa-0.7.0-alpha2	1.2	2.7	113	0.024	1	3	19	4	3	0	29
iReport-3.7.5	767.7	8001.8	2560	3.126	4	44	2316	206	5	4	2575
itext-5.0.3	183.1	2171.1	368	5.900	5	6	373	20	5	1	405
ivatagr.w.-0.11.3	0.2	2.4	132	0.018	1	20	2	4	1	2	29
jag-6.1	40.0	105.0	185	0.567	2	7	158	19	1	0	185
james-2.2.0	1.8	5.3	164	0.032	1	6	42	6	1	2	57
jasperreports-3.7.4	130.5	982.2	730	1.345	4	21	362	87	5	2	477
javacc-5.0	3.8	10.4	89	0.117	1	2	19	0	3	1	25
jboss-5.1.0	143.5	1121.4	1291	0.869	3	93	639	119	5	4	860
jchempaint-3.0.1	98.3	651.5	488	1.335	4	37	171	215	2	3	428
jedit-4.3.2	524.0	7554.0	922	8.193	5	14	1161	48	7	1	1231
jena-2.6.3	267.1	3532.1	699	5.053	5	21	620	116	2	4	763
jext-5.0	102.1	686.6	459	1.496	4	13	300	24	4	2	343
jfreechart-1.0.13	10.3	162.6	418	0.389	2	15	60	36	5	4	120
jgraph-5.13.0.0	62.3	366.7	331	1.108	3	8	216	11	4	2	241
jgraphpad-5.10.0.2	11.4	43.0	242	0.178	1	4	80	3	4	0	91
jgrapht-0.8.1	1.2	3.7	74	0.050	1	6	14	11	1	0	32
jgroups-2.10.0	50.2	268.5	506	0.531	2	7	269	39	2	2	319
jhotdraw-7.5.1	34.1	130.2	683	0.191	1	22	325	43	4	4	398
jmeter-2.5.1	148.9	2117.0	513	4.127	4	35	207	274	3	4	523

Continued on next page

Table 7.4 – continued from previous page

	Index					Architectural Smells					
						UD	CD		HL		Total
System	$\Sigma ASIS$	$\Sigma ASIS * w$	W	ADI	$q(ADI)$	Pkg	Cl	Pkg	Cl	Pkg	AS
jmoney-0.4.4	27.2	60.2	148	0.407	2	3	138	4	0	0	145
joggplayer-1.1.4s	12.1	30.4	209	0.146	1	3	96	2	3	0	104
jparse-0.96	9.6	43.1	58	0.743	3	1	30	2	1	0	34
jpf-1.5.1	0.9	2.2	72	0.030	1	2	18	2	1	0	23
jrat-1-beta1	19.2	100.3	176	0.570	3	19	69	46	1	2	137
jre-1.6.0	3136.1	81534.4	3997	20.399	5	145	5761	633	1	4	6544
jrefactory-2.9.19	129.4	1172.0	600	1.953	4	37	372	125	1	3	538
jruby-1.7.3	2586.6	128738.1	1904	67.615	5	46	3592	324	3	9	3974
jspwiki-2.8.4	87.4	711.7	313	2.274	4	17	181	78	4	2	282
jsXe-04_beta	32.8	135.0	286	0.472	2	7	145	8	6	0	166
jtopen-7.8	196.7	1790.9	618	2.898	4	3	613	4	1	0	621
jung-2.0.1	6.4	18.2	141	0.129	1	14	61	23	0	2	100
junit-4.10	6.4	20.3	122	0.167	1	11	45	22	1	1	80
log4j-2.0-beta	15.7	80.6	156	0.516	2	13	64	60	0	3	140
lucene-4.2.0	204.1	1150.6	1618	0.711	3	66	956	241	1	1	1265
marauroa-3.8.1	9.5	43.5	173	0.252	2	12	41	35	4	1	93
maven-3.0.5	19.5	205.1	180	1.139	3	27	28	102	0	6	163
megamek-0.35.18	551.8	7712.7	876	8.804	5	20	1106	72	3	1	1202
mvnforum-1.2.2-ga	60.3	376.9	349	1.080	3	26	201	56	1	2	286
myfaces_core-2.1.10	56.8	554.4	806	0.688	3	41	243	113	5	2	404
nakedobjects-4.0.0	111.9	758.3	766	0.990	3	99	369	279	0	2	749
nekohtml-1.9.14	0.5	1.1	43	0.027	1	2	5	2	1	0	10
netbeans-7.3	6898.9	49950.4	31592	1.581	4	1043	28865	2016	3	1	31928
openjms-0.7.7-beta-1	13.4	39.2	227	0.173	1	19	115	16	0	2	152
oscache-2.3	0.9	2.3	35	0.066	1	4	10	9	0	1	24
picocontainer-2.10.2	4.8	20.3	76	0.266	2	4	33	11	0	1	49
pmd-4.2.5	14.6	70.1	179	0.392	2	17	71	34	0	2	124
poi-3.6	195.1	1709.0	864	1.978	4	43	416	232	5	8	704
pooka-3.0-080505	437.7	6926.1	743	9.322	5	8	955	28	7	0	998
proguard-4.9	12.5	43.8	148	0.296	2	15	85	31	0	2	133
quartz-1.8.3	16.4	38.8	122	0.318	2	8	63	18	0	1	90
quickserver-1.4.7	18.1	55.6	137	0.406	2	6	95	14	0	1	116
quilt-0.6-a-5	4.6	13.8	31	0.447	2	3	25	2	0	0	30
roller-5.0.1	12.4	106.3	353	0.301	2	23	41	65	4	2	135
rssowl-2.0.5	834.5	10870.3	2116	5.137	5	51	2127	182	8	2	2370
sablecc-3.2	1.1	4.6	73	0.062	1	1	18	1	1	0	21
sandmark-3.4	52.2	227.1	394	0.576	3	23	238	22	2	0	285
spring-3.0.5	222.2	1472.0	1453	1.013	3	100	741	296	3	4	1144
squirrel_sql-3.1.2	23.3	52.6	116	0.453	2	2	106	2	0	0	110
struts-2.2.1	63.7	454.1	537	0.846	3	43	236	122	1	4	406
sunflow-0.07.2	50.6	364.4	146	2.496	4	7	115	38	1	2	163
tapestry-5.1.0.5	59.0	324.8	575	0.565	2	29	269	84	2	5	389
tomcat-7.0.2	107.7	646.0	796	0.812	3	35	452	92	4	1	584
trove-2.1.0	0.9	2.1	11	0.195	2	0	8	0	0	0	8
velocity-1.6.4	20.5	78.6	90	0.873	3	14	45	34	0	2	95
wct-1.5.2	18.9	168.9	268	0.630	3	35	81	69	1	3	189
webmail-0.7.10	6.2	44.1	109	0.405	2	7	20	6	3	1	37
weka-3-6-9	297.3	2109.3	1079	1.955	4	32	911	236	2	2	1183
xalan-2.7.1	423.8	4715.5	535	8.814	5	21	838	57	2	2	920
xerces-2.10.0	67.0	422.4	269	1.570	4	12	220	38	1	1	272
xmojo-5.0.0	0.1	0.1	5	0.022	1	1	2	0	0	0	3

7.4.3 *History*

The presence of an AS in the history of a project can have a different impact on the index. Figure 7.2 shows an example of project evolution through a graphical annotation. Version 1 (V1) is one version of a project, followed by Version 2 (V2). V1 has two architectural smells, such as: AS1 and AS2. AS1 has been deleted from the V2 of the project. AS2 is equal to the AS3 smell detected in V2. The comparison is made by comparing the subgraph (SG) involved by the architectural smells.

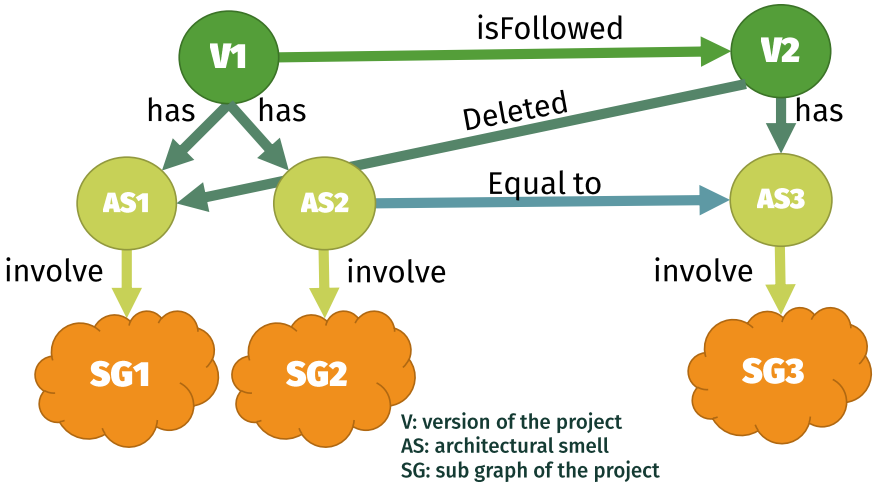


Figure 7.2: Example of AS evolution

$History(AS_k)$ is defined as the weight assigned to the trend of each type of smell as follows:

$$History(AS_k) = \begin{cases} \zeta & \text{If } AS_k \text{ has a Decreasing Trend} \\ \theta & \text{If } AS_k \text{ has an Increasing Trend} \\ \eta & \text{If } AS_k \text{ has a Stable Trend} \end{cases} \tag{7.5}$$

where the trends are evaluated as follows:

- Decreasing Trend: when the number of classes, files or packages involved in an AS is decreasing.
- Increasing Trend: when the number of classes, files or packages involved in an AS is increasing.
- Stable Trend: when the number of classes, files or packages involved in an AS is stable.

and where ζ, η and θ will be fixed according to the validation of the Index. The values are fixed as starting point at 0.5, 1 and 2 respectively.

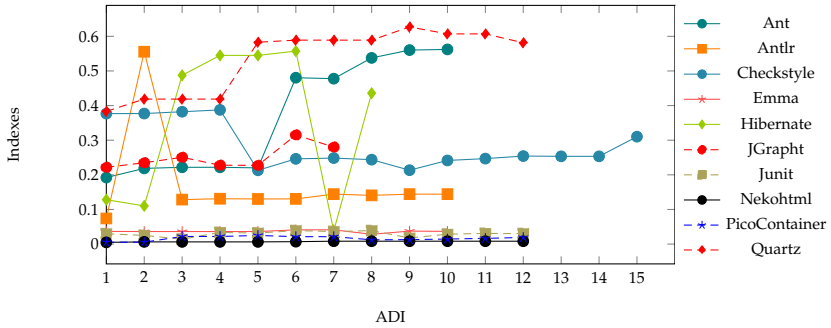


Figure 7.3: Evolution of ADI by project

7.4.4 *Architectural Debt Index evaluation*

This section is dedicated to the description of the study performed on more than 100 versions of 10 projects as outlined in Table 7.5.

Table 7.5: Projects selected for the ADI evolution evaluation

Project	Versions	Category
Ant	10	Tool
Antrl	10	Parser
Checkstyle	15	IDE
Emma	10	Testing
Hibernate	8	Database
JGrapht	7	Tool
Junit	8	Testing
Nekohtml	12	Parser
PicoContainer	12	Middleware
Quartz	12	Middleware

Figure 7.3 shows the evolution of the Architectural Debt Index (ADI) in the several major releases per project. The majority of the projects have an increasing trend (i.e. the projects are getting worst), since they did not take in consideration the ADI as quality index, so their ADI is growing over time, i.e., the ADI in the last version is higher than in the first version. Although, Checkstyle shows a decreasing trend of ADI before a major release publication and after that an opposite trend.

7.4.4.1 *Architectural Debt Indexes and SonarQube TDI: some remarks*

A trend analysis is performed on the Technical Debt Index provided by SonarQube and the Architectural Debt Index here defined. The evolution of the ADI has been compared with Technical Debt Index (TDI) of SonarQube, a well known index computed by SonarQube and explained in Section 2.5, and Figure 7.6 shows the comparison of the two indexes by project. The y-left-axe is referred to Technical Debt index and the y-right-axe is referred to ADI, but the attention is focused on the trends of the indexes since their

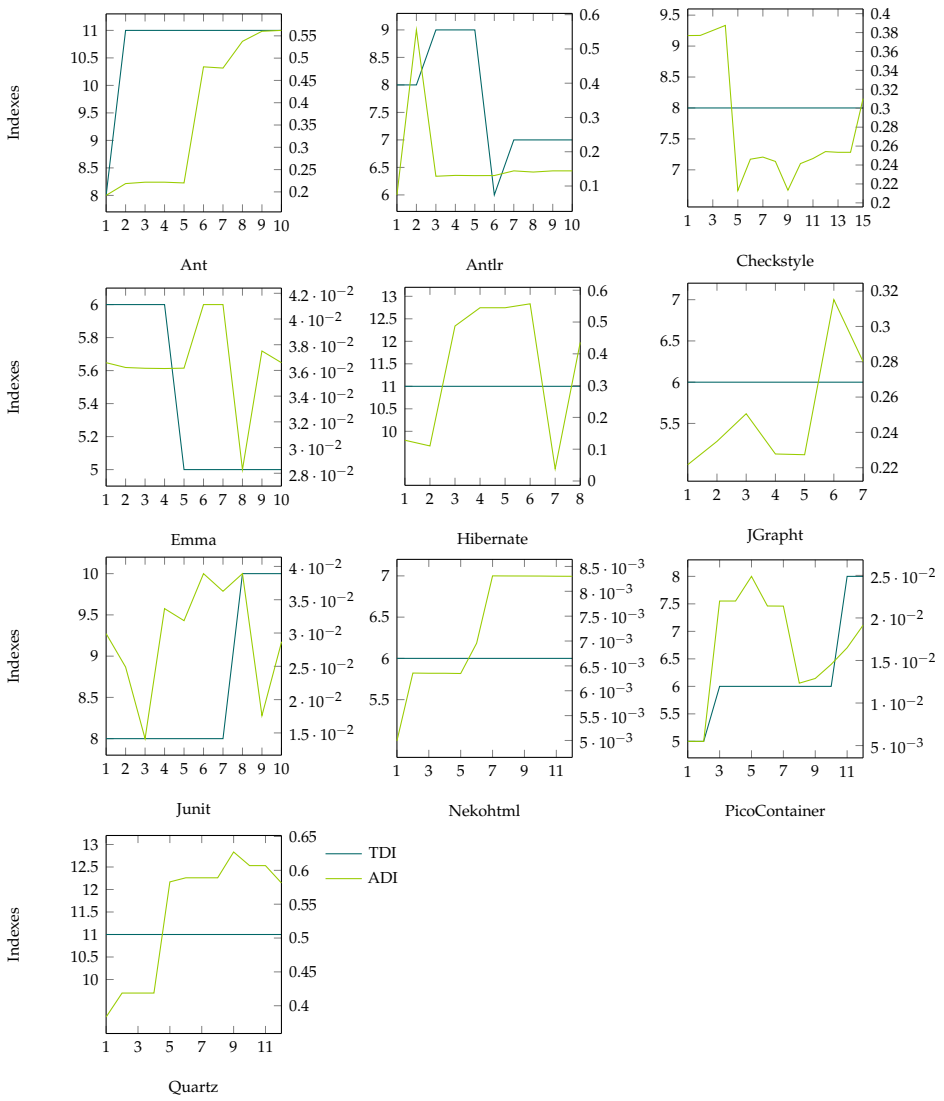


Table 7.6: Evolution of ADI and TDI Index by project

quantities/values have different scales. TDI and ADI are improved when it has an decreasing trend.

As showed in Figure 7.6, the trend of TDI is stable in 4 projects of 10, i.e., TDI has the same value at the first and last version; on the other hand, ADI has increasing trends for all projects. TDI is not improved in Ant, Junit and PicoContainer, and also ADI is grown worse. Moreover, ADI and TDI has same trend in Antrl and Emma projects, while ADI is getting slightly worst and TDI increasing in Ant projects.

ADI was computed on 10 projects and more than 6 versions per project, while TDI is widely used by the open-source community and computed through Sonarqube since it is one freely available tool that consider the evolution of the projects. The majority of the projects analyzed by using an implementation of ADI had an increasing trend and a worst overall software architecture quality, on the other hand the TDI had a stable or improved quality in contrast to ADI trends. The incongruity discovered among ADI

and TDI is related to their characteristics: ADI is focused is focused on the evaluation of *a)* the AS of the project and *b)* the AS evolution; TDI is focused on the detection of *a)* code and *b)* object oriented violations.

7.4.5 *Architectural Debt Index Profiles*

The ADI index could be computed according to different quality attributes (defined in ISO/IEC 25010:2011 [65]). Examples of quality attributes could be:

- *Performance efficiency*: The performance relative to the amount of resources used under stated conditions;
- *Security*: The degree of protection for information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them (e.g., Ambiguous Interface smell [52]);
- *Reliability*: The degree to which a system or component performs specified functions under specified conditions for a specified period of time.

Different AS have been defined in the literature, and new ones can be defined. It could be interesting to identify some categories of AS that could have an impact on a specific quality attribute. Obviously, the same AS may have an impact on more quality attributes, hence the categories are not disjoint with respect to the AS they group. For example, the Ambiguous Interface smell defined by Garcia et al. [52] can be included in the security category since it represent an interface that provides a single general entry point to access into a component.

If categories are identified and associated with one or more AS, and if a tool (e.g., Arcan) is able to detect these AS, it is possible also to define different profiles for the Architectural Debt Index.

For example, if it is necessary to evaluate the ADI with respect to a profile, e.g., Performance, the tool computes the index and provides an evaluation considering only the AS belonging to the Performance category. Moreover, a AS belonging to the Performance category could have as output for its ASIS score a value between the range from 1 to 5 using the function $q()$, as defined below:

$$q(ASIS_k^t) = \begin{cases} 1 & \text{If } 0.00 \leq quantile(ASIS_k^t) \leq 0.20 \\ 2 & \text{If } 0.20 < quantile(ASIS_k^t) \leq 0.40 \\ 3 & \text{If } 0.40 < quantile(ASIS_k^t) \leq 0.60 \\ 4 & \text{If } 0.60 < quantile(ASIS_k^t) \leq 0.80 \\ 5 & \text{If } 0.80 < quantile(ASIS_k^t) \leq 1 \end{cases} \tag{7.6}$$

where $quantile(ASIS_k^t)$ is the quantile associated to the $ASIS_k^t$ computed for the AS_k^t of type t in the reference dataset, e.g., given an AS and its ASIS score

with $q()$ of 5: it should be highlighted in the output as very bad architectural smell in the associated profile. The results obtained through the $q()$ function for all the architectural smells belonging to a category should be aggregated using the *max* operator, to obtain the highest value of the set, e.g., As_i and AS_j, belonging to the Performance profile, are architectural smells with $q(ASIS)$ of 5 and 2 respectively, the aggregated value for their profile is 5.

Moreover, both the discretization of the ASIS, the profile association and ASIS value aggregation could help in the results inspection and refactoring estimation effort. Another potential application of these different ways of composing the ASIS lies in the definition of quality compliance levels. In fact, the quality compliance level could be setup for each profile, or the minumum quality for each profile can be set, by defining the maximum ASIS level for all profiles. These factors can be combined with the global ADI of the project to have finer-grained definition of the desired quality of the overall project.

7.5 CONCLUSION

In this chapter a new index oriented to the evaluation of architectural issues as AS is proposed and integrated in the Arcan tool. Severity of all the architectural smells of Arcan are given and explained. A validation of ASIS with Arcan has been performed on a dataset of 109 open source projects. The evolution of the ADI and a comparison with SQ TD index has been performed on 10 projects considering more than 100 versions on total.

In the following, answers are provided to the questions posed at the beginning of this chapter.

WITH RESPECT TO RQ1: *How should a new index be formulated to better represent the architectural debt?* The defined index is composed by the summation of the value of the $ASIS$ for each AS times the weight of the AS over the total number of dependency involved in an AS, times the trend estimated of the index, called *History*. The $ASIS$ estimates the *cost-solving* of an architectural smell through the computation of its *SeverityScore* and its *PageRank*, as explained in Section 7.4.1. A proposal has also been described to compute ADI on different profiles, based on the ISO/IEC 25010:2011 standard, where the architectural smells are grouped according to the affected quality attribute, as explained in Section 7.4.5. Moreover, the ADI can be used to identify the most critical classes or packages in the projects, in this way the developer/maintainer can easily identify and focus his attention on the most critical classes or packages.

WITH RESPECT TO RQ2: *How can we estimate the severity of an Architectural Smell?* It is possible through the definition and the detection of the architectural smells. For example, the *SeverityScore* of the Hub-Like Dependency smell is given by the total number of dependencies among the class c (or the package) affected by the HL smell and the classes (or packages) which call or are called by c. This is strictly related to the *cost-solving* of the architec-

tural smells. In the same way, the Severity Score for the other architectural smells were defined in Section 7.4.1. The *PageRank* was identified as a smell-agnostic way to weight the severity of an AS, since it indicates the most important and popular place in the dependency graph (i.e., the elements of the dependency graph that are the hardest to refactor). With agnostic, we mean that the *PR* uses the dependency graph and it is independent from the AS type.

WITH RESPECT TO RQ3: *How does the evolution of the new index behave through the analysis of projects history? and how does it behave with respect to another index?* The majority of the projects have an increasing trends of ADI showing that the internal architectural quality is getting worst over time, as shown in Figure 7.3. Moreover, the TD index showed different trends respect to the ADI highlighting weakness of available indexes on architectural quality evaluation. Although the ADI got worst the TDI was stable or getting better in the majority of the projects. Moreover, the incongruity discovered among ADI and TDI is related to their characteristics: ADI is focused on the evaluation of *a)* the AS of the project and *b)* the AS evolution; TDI is focused on the detection of *a)* code and *b)* object oriented violation.

In the future, the index will be evaluated on a large dataset of both open source and industrial projects, with the aim to get the feedback of the developers. Hence, the weights assigned for example to the History could be changed according to these validations.

On the study regarding the ADI evolution I co-supervised the bachelor thesis of Matteo Marchesi and on the definition and implementation of the ADI, I will submit a paper entitled "A new Architectural Debt Index" to the next Technical Debt Conference 2018, Co-located with ICSE Conference.

8

ARCHITECTURAL SMELLS REFACTORING: A PRELIMINARY STUDY

Refactoring has been largely studied in the literature in several directions and by considering different features, techniques and its impact on different quality issues [110, 111, 125]. Many works have been done in particular on code refactoring, with often a focus on code smells refactoring, as for example [25, 165].

As outlined by Zimmermann [167] given the success of code refactoring, it is surprising that architecture refactoring has not received a great attention yet. Architecture refactoring is required to maintain the structural quality of any complex and evolving project and requires a significant amount of effort and time compared to code refactorings [149].

Architecture refactorings helps architects to identify potential problems in a software architecture and identify the refactorings steps to solve them. An example of these problems is given by the architectural smells. Identify these problems is not an easy task and hence the availability of tool that support the architectural smells detection is particularly relevant for architecture refactoring. Hence in this context, architecture refactoring can be seen as a set of architectural activities devoted to the removal of one or more architectural smells. Moreover, through the refactoring of architectural smells, one could expect an improvement of some quality issues. Establishing and evaluating the benefits of refactoring, as also outlined by Ó Cinnéide et al [119], is certainly quite unclear and not an easy task, due to the intrinsic difficulties and trade-off to be faced during the refactoring of the smells.

Removing an architectural smell is a task that could involve the choice among different refactoring steps, and this choice can lead to a different impact on the quality issues. Moreover, it could be very difficult to remove them in some case, owing to too much conflicts to be solved. It would be interesting to identify high-impact refactorings [30], that are the refactorings with a strong impact on the quality of the project's architecture.

In this chapter, the attention is focused on the detection of three different architectural smells (AS) through Arcan [17] (Chapter 4). Cyclic Dependency smell, Unstable Dependency and Hub-Like Dependency smells are detected. The AS are detected and all removed on three open source projects of different size. Moreover, the effect of the refactoring of these smells is analyzed on some quality metrics. In particular, some metrics and structural quality indexes computed by two known commercial tools were considered. The refactoring is performed by removing first all the AS of one type for all the three different AS, then by removing all the AS found in the analyzed projects according to the choices taken in the different refactoring scenario, that we describe in details.

Through this experimentation, I aim to answer the following Research Questions:

RQ1.1 How often can refactoring opportunities and/or recommendation be easily identified for architectural smells?

RQ1.2 Can refactoring patterns be discovered that can help in the removal of an architectural smell?

RQ2.1 Which is the impact of the refactoring of the architectural smells on the considered quality metrics?

RQ2.2 Can architectural smells be identified with a higher impact (positive or negative) on the quality metrics?

The Chapter aims to outline and identify the major difficulties in the refactoring of the three architectural smells and their impact on some quality metrics through the answers to these RQs. In particular, the answers to *RQ1.(1-2)* aim to analyze whether it is difficult or not the refactoring of architectural smells, having previous experience in the refactoring of code smells [13]. The analysis consists in finding and proposing some refactoring recommendations for the considered architectural smells. Moreover, through this study different insights about the features and difficulties of architecture smells refactoring are outlined. The faced problem is related both to the possibilities of identifying common refactoring steps to remove an AS and to the possibilities of identifying the best sequence of AS to be refactored in order to improve a project.

RQ2.1 aims to analyze the impact of the AS refactoring on quality issues, to get some indication if their removal effectively improve software quality, according to the considered metrics/quality indexes. The impact on software quality metrics not used for the AS detection is outlined and analyzed. Strictly related to the previous question, through the answer of *RQ2.2* aims to identify the AS which has a higher impact on the software quality metrics. This information could be used to identify some prioritization on the more critical smells to be removed first.

8.1 SET UP OF THE STUDY

8.1.1 *Analyzed Projects*

Table 8.1 shows the size of the systems that we have considered in this study. These projects are taken from the Qualitas Corpus [154] and are briefly described below.

Commons-collections[1] is a library developed by Apache Foundation. The aim of this library is to provide powerful data structure in order to accelerate the development of Java applications. It provides several data structures implementation, such as Map, Set, List and many different implementation of Iterators.

1 http://commons.apache.org/proper/commons-collections/

Table 8.1: Analyzed Projects

Project	Version	LOC	Methods	Classes	Packages
Collection	3.2.1	63926	6658	273	12
Jag	6.1	28124	1434	121	16
Informa	0.7.0	29330	1614	150	14

Jag[2] is a tool project developed by Finalist IT Group to create complete J2EE applications. Jag allows to easily integrate different libraries and frameworks in web application projects and offers solution to common problems triggered by web application development.

Informa[3] is a news aggregation library, commonly used to obtain news from different channels and protocols, e.g., using ATOM, RSS, RDF. It could be re-used in other projects, e.g., search engine and it could export data in different way, such as, XML, Hibernate.

8.1.2 *Data collection*

To evaluate the impact of architectural smells refactoring we have considered some structural quality indexes evaluated through Sonargraph and Structure101 tools. We decide to compute these indexes since, see Section 7.1, we evaluated indexes provided by different tools and we found the ones computed by the above tools particularly useful to assess architectural issues. I have not considered in this study the ADI index which I defined in the previous chapter, since the index has not been externally validated yet.

Sonargraph[4] allows to compute a Structural Debt Index and two types of metrics called Structural Erosion (Type Dependencies and Reference) described below. Sonargraph's Structural Debt [112] is quantified through two measures (see Section 2.5): *Structural Debt Index* (SDI) and *Structural Debt Cost* (SDC).

Structural Erosion - Type Dependencies on project level: this metric calculates for the entire project the number of type dependencies that must be cut in order to remove all package cycles in the project (this is not always the exact minimum number, but a heuristic-based guess).

Structural Erosion - Reference on project level: this metric is equal to the previous one, but it considers the number of references to be removed.

While Structure101[5], shows a *Structural over-Complexity (SoC)* view to estimate the percentage of the project involved in architectural issues and it displays two measures: %Tangle and %Fat (see Section 2.5).

2 http://jag.sourceforge.net/
3 http://commons.apache.org/proper/commons-collections/
4 https://www.hello2morrow.com/products/sonargraph
5 https://structure101.com/

We outline that we decided to not consider metrics, as LCOM, CBO, FAN-IN, FAN-OUT [38], and dependency metrics as Distance from the main sequence, Instability, Abstractness, Afferent and Efferent Coupling [109], since these metrics have been used to detect the AS through Arcan, and an impact on these metrics can be obviously expected.

8.2 ARCHITECTURAL SMELLS REFACTORING RESULTS

In this refactoring study through the analysis of the three projects described in Section 8.1, we followed the steps described below. We use the word *base* to identify the project before any kind of refactoring step:

1. we detected through Arcan the three types of AS (UD, CD and HL) on the three projects;
2. we removed all the AS of one type from all the projects, in the cases where we could not remove all of them we motivated the reason; we did the same for all the three types of AS (UD, CD and HL), starting for each AS from the *base*, as following:
   - *base* → (UD refactoring) → *UD-Refactored Version*
   - *base* → (CD refactoring) → *CD-Refactored Version*
   - *base* → (HL refactoring) → *HL-Refactored Version*
3. we checked the impact of the refactoring on the number of AS found in the three projects, in order to check if by removing the AS of one type, we removed also other types of AS;
4. we removed all the three types of AS, or part of them, in the three projects. To decide the order of the AS to be removed first, we took into account the AS that at step 3 lead to the removal of the higher number of AS;
5. we checked the impact of the refactoring on some quality indexes values;
6. after the refactoring of the smells, both at step 2 and step 4, we tested the projects according to their software test suite in order to check if the refactoring steps did not introduce functionality problems.

In Table 8.2 we report the number of AS detected and removed in each project. In the *Base* line we outline the number of AS detected before the refactoring, and in the other lines the number of AS after refactoring. For example, the *HL* line for the Jag project corresponds to the refactoring of the HL, and we see that the number of UD remain the same (16), the number of CD at class level increases (from 8 to 9), the number of CD at package level remains the same (3) and the number of HL go to 0. We also outline that for the CD smell at class and package level, we show also the number of classes and packages involved (i.e. 9/87: 9 CD at class level respect to 87 classes involved in at least a CD).

Table 8.2: Overall architectural smells in the analyzed systems before and after the refactoring steps

Refactoring	UD	CD		HL
	Package	Cycle / Class	Cycle / Package	Class
		Jag		
Base	16	8 / 81	3 / 12	1
CD	16	7 / 77	3 / 12	1
HL	16	9 / 87	3 / 12	0
UD	7	8 / 81	2 / 6	1
All AS	7	7 / 66	3 / 12	0
		Common-collections		
Base	16	16 / 53	1 / 8	1
CD	16	2 / 7	1 / 8	1
HL	16	11 / 39	1 / 8	0
UD	14	10 / 35	1 / 8	1
All AS	14	4 / 15	1 / 8	0
		Informa		
Base	0	2 / 6	1 / 3	3
CD	0	0 / 0	1 / 3	3
HL	0	2 / 6	1 / 4	0
UD	–	– / –	– / –	–
All AS	0	0 / 0	1 / 3	0

8.2.1 *Refactoring Results of the Hub-Like Dependency (HL) smell*

According to Table 8.2 we see that we found 1 HL in Jag and Common Collection projects and 3 in Informa. We removed all the Hub-Like from the three projects.

We have not found false positive instances. However, we figured out which are the conditions where a Hub-Like could be a false positive instance. Hub-Likes are highly used classes of the project. We don't know if those classes are used from other projects, since here we analyzed one project at time. However, we know if the class uses external classes of the project, e.g., classes of the package `java.util.*`. If the class has the majority of outgoing dependencies related to a project library, it should not be considered a Hub-Like. Classes of this kind are simple, because they use default functionalities (e.g., list), but are frequently used and implement the main functionalities of the project.

By removing this type of AS in Informa, the number of CD smells at package level remain the same, instead the number of CD at class level increases by one because we broke a chain cycle in two cycles by delegating responsibility to other classes. However, we experienced that the Tiny Cycle Dependency smells (see Figure 1) were participating to make Hub-Like bigger, increasing both Fan-In and Fan-Out. If the classes of these Tiny Cycle Dependencies are anonymous or nested, they should not be considered at all or be weighted less in the computation of the Fan-In and Fan-Out metrics used for the detection of this smell.

To remove the HL smell we usually applied these refactoring steps:

- Extract super class: we moved fields and high frequently used methods to a super class, in order to reduce the size of in-coming and out-going dependency.
- Move method: we looked for opportunities to move methods to other classes or to move the logic itself.

The main problem for the refactoring of Hub-Like is not related to the study of out-going dependencies (Fan-Out) or in-going dependencies (Fan-In), but the main problem is to understand the causes of the architectural smell, e.g., the nature of the dependencies for the affected classes.

8.2.2 *Refactoring Results of the Unstable Dependency (UD) smell*

According to Table 8.2 we found 16 UD in Jag and Commons-collections, none in Informa.

We removed 9 UD from Jag and 2 from Common-collections. In order to remove the architectural smell it was necessary to move misplaced classes from its package to the sub-package where they are used.

We have not removed all the UD instances since in some cases the packages considered as UD were only dependent of main packages, such as package where all the others subordinate packages are placed. We noticed some false

positive instances in the cases of packages with UD having a medium value for the Instability metric (i.e. near to 0.5), since these packages were related to packages slightly more unstable. The Instability metric is used in the detection of the smell.

By removing this type of AS, we removed also 6 CD at class level in Common-collections and 1 CD at package level in Jag. The other smells remain the same.

To remove the UD smell we usually applied these refactoring steps:

- Move class: we searched for those classes which were using resources of the package with Unstable Dependency. If one class is used by other classes in the package, we had to move the related classes to the same package to remove the architectural smell.
- Merge packages: we removed packages that were left with only few classes (i.e. less than 5) and we merged them with the package with the Unstable Dependency.

8.2.3 *Refactoring Results of the Cyclic Dependency (CD) smell*

According to Table 8.2 we see that we found 8, 16 and 2 CD at class level and 3, 1 and 1 CD at package level respectively for Jag, Common Collection and Informa projects.

We have not removed all the CD instances since in some cases cycles were a right consequence of a correct construction and the involvement of design patterns.

Cyclic Dependency smells related to GUI are usually related also to the Model View Controller pattern, but they have to considered as false positive instances. We noticed that Cyclic Dependencies were in classes using the GUI library provided by Java which requires the generation of anonymous classes instances. We removed cycles generated in this way where it was possible because the anonymous classes were not using resources of the creator class and we moved the logic of the called method to the anonymous classes. Moreover, the method body called by the anonymous classes was sometimes empty, so we removed all these methods.

Moreover, we found that CD is frequently related to Factory Method design pattern, and it is hard to remove CD involving this kind of design pattern.

By removing this type of AS, the number of UD and HL smells remain the same. To remove the CD smell we usually applied these refactoring steps:

- Move method: method moved from the declaring class to the using class. If the method is wrong-placed and create a Cyclic Dependency, we need to replace them to correct class. This is usually the same class where it is called the method.
- Extract super class: Class with too many responsibilities should be redesigned and a super class is usually created with the common used

methods and fields of the class. This represents also an opportunity to improve readability of the code and its maintainability.

- Move class: Nested classes that are used also from outside the class, should be extracted as class stand alone. This is important to avoid duplicated nested classes in the project and improve in this way reusability.

Package cycles involve a large number of classes of the project in general. It was unmanageable to remove package cycles in the analyzed projects. For example, Informa package cycle involves 66 classes over three packages, showing that the cycle among these three packages have a strong relation among each other.

8.2.4 *Refactoring Results of all the three AS*

At this step, we tried to remove all the types of AS found in the three analyzed projects, according to step 4 described at the beginning of Section 8.2.

We defined the Best Impact Refactoring Architectural Smell sequence (BIRAS sequence) according to Table 8.2. The BIRAS sequence prioritize the refactoring action which reduced more architectural smells in the project, as following:

- **Jag:**
  - $UD \rightarrow CD \rightarrow HL$
- **Commons-collection:**
  - $UD \rightarrow HL \rightarrow CD$
- **Informa:**
  - $HL \rightarrow CD$

We achieved the goal to reduce the amount of architectural smells in any project. Informa was the project with the lower number of architectural smells. We obtained the best results by removing 5 architectural smells (2 CD at class level and 3 HL) and we left only one CD at package level.

Common-collections was the project with the highest number of architectural smells. We removed more cycles in Commons-collection because they were related to a low number of classes. We had more problems with Jag because it was affected by a greater number of classes and packages in cycles than the others projects. For example, `JagGenerator` is a lead class of Jag that uses and controls many other classes. We had several problems to manage the refactoring of this class also because it was involved in several architectural smells.

The number of Unstable Dependency was not modified by the refactoring of Hub Like and Cyclic Dependency in any project. Indeed, Unstable Dependency is constant at zero for Informa and it was stable to the value reached after the refactoring of the other projects. Moreover, the number of package cycle is not modified by any refactoring actions in any project.

8.2.5 *Impact of Refactoring on Quality indexes*

We checked the impact of the AS refactoring on the metrics/quality indexes introduced in Section 8.1.2.

Table 8.3 reports the values of the metrics computed by Structure 101, related to the percentage of the project involved in a Tangle and exceeding the Fat metrics, for the base version of the projects and the final version after refactoring.

Tangle and Fat metric percentages slightly increased for Jag and Commons Collections, while Informa has a stable percentage of Tangle and a decreased percentage of Fat metric.

Table 8.3: Structure 101 - Structural over-complexity index components

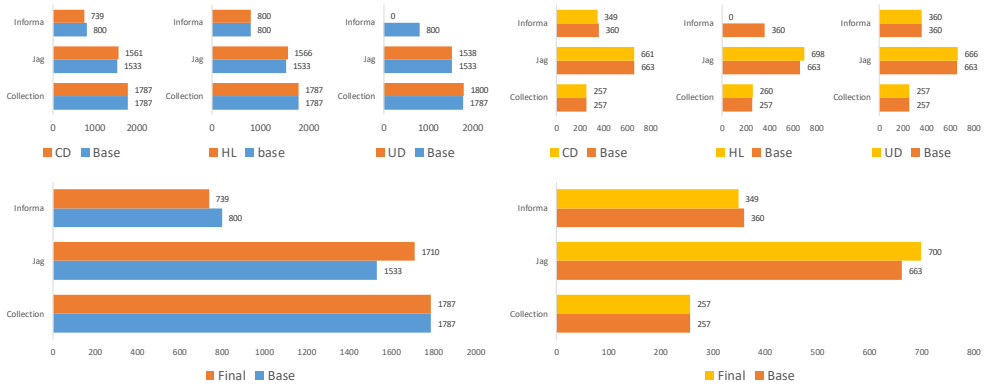
	Jag		Commons Collections		Informa	
Refactoring	%Tangle	%Fat	%Tangle	%Fat	%Tangle	%Fat
Base	90,909	20,960	50,251	92,000	1,508	83,920
Final	92,462	21,610	50,754	92,960	1,508	78,992

Sonargraph indexes presented at plots in Figures 8.1a-8.1b-8.1c show different results. The impact of refactoring on Informa measured by this indexes shows decreasing values so the final version of this project has been improved from the base version according to Sonargraph indexes. Commons Collections and Informa show different results. Commons Collections has stable results between base and final version. While, Jag has increasing indexes values in the final (refactored) version.

8.3 DISCUSSION

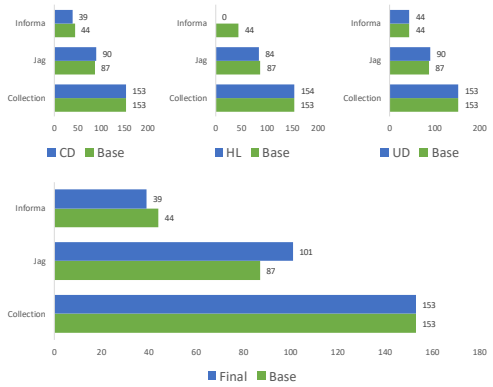
8.3.1 *Hints on how to refactor the AS*

We used all the feature of Arcan in order to understand and solve all the architectural smells discovered by the tool. The output files of Arcan were useful for us for a quick evaluation of the results, but we exploited the graph database to obtain more details about the AS. The graph database was particularly useful to inspect the dependency graph in order to understand the architectural smells dependency with affected packages or classes. For example, we inspected outgoing and ingoing dependency of Hub Like smell to discover refactoring opportunities, such as delegating some functionality to classes most related with the class affected by the Hub Like in order to reduce Fan In and Fan Out metrics.



(a) Structural Debt Index

(b) Structural Erosion Reference



(c) Structural Erosion Type Dependency

Figure 8.1: Sonargraph indexes

8.3.2 *Hints on how to improve the AS detection through Arcan*

During this experimentation we found different cases of false positive instances, and these can be seen as hints on how to improve the AS detection of Arcan. In particular we found the following false positive instances:

- The detected false positives for Hub Like Dependency smell reflect abstract classes, interfaces and in some cases classes which implement the Singleton design pattern [50], since by design are expected to be reused in several parts in the project. However, as explained in Section 8.2.1, if the class uses only external resources like `java.utils.List`, it should not be considered a HL smell because it is using java standard library.

- The Cyclic Dependency smell false positives reflect classes which implement Factory Method and Model View Controller design pattern and nested (hidden) classes, as explained in section 8.2.3. Moreover, we can find that some AS have not to be removed, e.g., some cycles seem to not be breakable because of their semantics, as also outlined by Laval et al. [82], since engineers can organize classes in multiple

small packages instead of one big, where package cycles in this reorganization are not critical for the modularity of the application.

- The Unstable Dependency false positives are packages which are dependent to slightly more unstable packages. We could implement some filters to exclude these false positives.

8.3.3 *Hints on which AS remove first*

According to the experimentation described in this work and one done with the developers of two projects related to the validation of Arcan results in terms of precision and recall measures [19], we decided to define the Severity of architectural smells, see Section 7.4.1. The Severity will allow to prioritize the more critical AS smells, allowing to save effort during the refactoring process.

Moreover, during refactoring we can focus the attention on the classes or packages involved in multiple AS. It could be useful to consider first these ones, since they are certainly more critical and by removing an AS we probably could remove other related AS, where for related we mean those AS present together. For example, Hub Like smell is frequently related to Cyclic Dependency and Unstable Dependency could be related to Cyclic Dependency between Package.

8.4 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of this study.

8.4.1 *Threats to Internal Validity*

Threats to internal validity may arise from two main sources: the detection of architectural smells and the refactoring choices.

For architectural smell detection, we apply Arcan, see Chapter 4, and currently able to detect different architectural smells. Basing our analysis on the output of a tool may bias our results in different ways: *1)* the tool may not support the detection of all architectural smells, and *2)* the tool may suffer from a detection error, missing some actual architectural smell instances. In the first case, we may work on mostly non-relevant architectural smells (because the relevant ones are not supported by the tool), while in the second case we may miss relevant architectural smells in our refactoring. Architectural smell detection is a recent an active research area; different architectural smell definitions have been proposed, but many of them are not currently detectable by any tool, and very few tools are available in general, as largely outlined in this thesis. In our tool, we support some of the most known architectural smells, which we consider among the most relevant. In this way, we mitigate threat *1*: while some interesting architectural smells may not be supported by Arcan, the currently supported ones are among the most important we are aware of. As for threat *2*, we conducted a review of Arcan

results with external developers, including the maintainers of the analyzed projects, with very good results [19]. This mitigates threat 2.

The other main threat to the internal validity of our work is the application of refactoring to the detected smells. In our work, refactoring decisions are made by the developer inspecting Arcan results. This may lead to subjectivity in the choice of the refactoring to be applied and in the way of applying it. For this reason, together with on Ms Student and a Post Doc fellow, I independently inspected Arcan results, choosing a refactoring for the detected smells. Then, when choices were different, they discussed their choices to reach an agreement case by case. The execution of each refactoring has been performed by my self, and reviewed by the other two to avoid incorrect or clearly sub-optimal choices. In this way, we mitigated the risk that our refactorings may be improved by other developers, or may not be representative of the choices made by other developers.

8.4.2 *Threats to External Validity*

Threats to external validity are related to the ability to generalize the obtained results. We performed our analysis of a restricted set of Java projects. Therefore, we cannot claim that our results can be generalized outside the scope of these projects or our development team. In addition, our results do not point to a clear effect of architectural smell refactoring on architectural measures. Additional analyses (and/or replication) should be performed independently to confirm our results.

8.4.3 *Threats to Conclusion Validity*

Threats to conclusion validity are related to factors that lead us to take the wrong conclusion while looking at our results. The architectural measures we applied to compare the state of the projects before and after the refactoring have been extracted by established commercial tools, which define their measure publicly. This ensures both good quality in the extraction of the measures and their verifiability. On our side, when applying the tools to the projects, we checked the tools settings before applying them, to avoid distorting the obtained measures through an incorrect application of the tools. In our results, we found that these measures do not uniformly or significantly reflect the changes we made to the projects. In our opinion, this should not be attributed to measurement errors, but to the fact that the measures do not reflect the kind of changes we made to the projects.

8.5 CONCLUSIONS REMARKS

In this work through our experimentation, we provided some insights about the characteristics of architecture smells, their removal and their impact on some quality metrics/indexes. Insights that could be useful for future investigations and development of AS detection and refactoring tools.

We can observe that it's not easy to remove the AS, this task is certainly more complex than code smell refactoring. Moreover, it's not easy to evaluate the impact of refactoring. We considered the impact of refactoring on the value of some quality metrics/indexes computed by well known tools and we analyzed the different scenario that we had to face. Our aims with respect to this point, was to observe if the AS refactoring is reflected also in the improvements of some quality indexes values. From our results this was not observed. Effectively evaluating the benefits of AS refactoring requires further research and refactoring in general, not focused only AS, has been recently considered an open issue by other authors [119].

We provide below the answers to our research questions in this context:

RQ1.1 *How often can refactoring opportunities and/or recommendation be easily identified for architectural smells?* From Table 8.2, we can see that we could always remove HL instances, while we never removed package cycles. Removal was possible instead for most class cycle (except for Jag). For UD, only less than a half of the instances have been refactored. Summarizing, refactoring opportunity easy depend a lot on the type of the AS, and on its size: wider AS, like package cycles are much more difficult to control, and therefore to refactor.

RQ1.2 *Can we discover some refactoring patterns that can help in the removal of an architectural smell?* In Sections 8.2.1, 8.2.2, 8.2.3 we identified a series of refactoring steps that can be recurrently applied to each kind of AS to remove it. Most steps involve moving program elements to a different container (class or package), to redistribute dependencies, but in specific cases splitting classes is a viable solution to distribute homogeneous dependencies sets.

RQ2.1 *Can refactoring patterns be discovered that can help in the removal of an architectural smell?* We considered metrics extracted by two tools, i.e., Structure 101 and Sonargraph. By comparing the values of the extracted metrics, we could not find recurrent changes in one or more metrics for the same kind of refactoring. Instead, all Sonargraph metrics change in the same way on the same project and the same refactoring set, with the only exception of UD in Jag. Structure 101 metrics got slightly worse after refactoring. Both Sonargraph and Structure 101 metrics changed very lightly in response to our refactoring operations, meaning that they probably consider aspects of the architecture that are not affected by architectural smells.

RQ2.2 *Can architectural smells be identified with a higher impact (positive or negative) on the quality metrics?* No, it was not possible to observe a single set of refactorings having a significantly large effect on the values of the considered metrics. In any case, we have to observe that we defined and applied the Best Impact Refactoring Architectural Smell sequence (BIRAS sequence) described in Section 8.2.4.

9 CONCLUSIONS AND FUTURE DEVELOPMENTS

In this thesis I discussed the need of identify and evaluate the architectural erosion of software applications through the support of architectural smells detected by Arcan. Moreover, I studied the architectural smells evolution, their prediction and the possible correlations between architectural smells and code smells. A new architectural debt index is proposed in order to estimate this architectural erosion.

9.1 CONCLUSIONS

In summary the contributions of this research are the following:

- *Architectural smell detection through Arcan (Chapter 4)* – I developed Arcan, a tool of architectural smells detection on Java projects. The tool relies on the dependency graph to represent the extracted information. I defined the techniques to detect six architectural smells, where one architectural smell is detected using the historical development data. Arcan detects two new architectural smells: Multiple Architectural Smell and Specification-Implementation Violation. The detection of the other architectural smells is enhanced w.r.t. previous approaches. The dependency graph provided by Arcan is useful to identify the possible refactoring opportunities of an architectural smell and the metrics used in the detection techniques of the architectural smells can be exploited to identify the most critical ones (i.e, architectural smells severity as explained in Chapter 7.
- *Evaluation and validation of Arcan results (Chapter 5)*
  - *Initial evaluation of Arcan results (Section 5.1)* – The first evaluation of Arcan results was performed on 7 open source projects and a preliminary comparison with the results of other two tools [17].
  - *Architectural smells validation: an industrial case study (Section 5.2)* – The second evaluation of Arcan results carried out with real-life software developers in a industrial case study [19]. Two different projects were analyzed by Arcan and a precision of 100% was observed, since Arcan found only correct instances of architectural smells.
  - *Architectural smells validation: a mixed method approach (Section 5.3)* – The evaluation of the tool results through extensive mixed-methods research, that shows that the tool is *precise*, since it detects 100% of the architecture smells effectively present in the analyzed source code, and is reliable, since it detects the *correct* architecture smell in over 84% of the cases (submitted [135]).

- *Empirical analysis (Chapter 6)* –
  - *Architectural smell prediction and evolution (Chapter 6.1)* – I applied machine learning techniques to predict architectural smells based on historical smell information and I studied the evolution of the architectural smells[137]. I discovered that historical architectural smells and changes done at package level can be used to predict the presence of architectural smells in the future. The architectural smells are visually correlated and generally increase over time.
  - *Are architectural smell independent from code smells? (Chapter 6.2)* – I conducted a large-scale empirical study investigating the relations between code smells and architectural smells and if code smells affect the presence of architectural smells and vice versa. I found empirical evidence on the independence between code smells and architectural smells and therefore, I can suppose that the presence of code smells does not imply the presence of architectural smells, (submitted [20]).
- *Proposal of a new architectural debt index (Chapter 7)* – a new index (ADI) oriented to the evaluation of architectural issues as AS has been proposed. I evaluated and integrated the ADI in Arcan through the definition of the severity of architectural smells (e.g., *SeverityScore*).
- *Architectural smell refactoring (Chapter 8)* – I presented a preliminary study conducted on selected projects applying refactoring actions in order to remove the detected architectural smells. I detected refactoring steps applicable on all types of considered architectural smells.

In conclusion, the main findings of the thesis are related to the implementation of new techniques for architectural smells detection through the Arcan tool. I have discovered and proved the independence of code smells from architectural smells, highlighting the importance of the architectural smells detection since they could point out problems not discovered through code smells. I proposed a new architectural debt index focused on architectural smells detection, their severity (*SeverityScore*) and their impact on the system (*PageRank*). The index can be used to compare projects, their evolution and to estimate the cost keeping of the debt. Moreover, the importance of the architectural debt index (ADI) is related also to architectural smells refactoring, since it is not easy to estimate the cost to remove architectural smells, this cost could be evaluated by considering the architectural smells severity computed through the ADI index.

9.2 FUTURE DEVELOPMENTS

- *Architectural smells detection* – Arcan currently detect six AS, it would be interesting to detect other architectural smells, also not strictly related to dependency issues, and further investigate and validate the two new architectural smells defined in Chapter 4 (e.g., Specification-Implementation Violation and Multiple Architectural Smells). In the

future, further releases of the tool are planned to: (a) extend programming language support; (b) improve the visual representation of Arcan results.

- *Architectural smell prediction and evolution* – In future developments, I would like to extend this study by analyzing more projects, also in other domains, and to extend the analysis to a larger set of lag settings, to have a more precise view of the prediction performances further in time. I are particularly interested in understanding and studying the co-evolution of architectural smells. This is particularly useful to better understand the AS and their evolution, but also their removal. Moreover, I would like to extend the study according to the prediction of changes through AS in history, to check if the presence of architectural smells in the projects evolution can be used to predict software changes. Another direction of investigation is related to a deep study on architectural smell false positives identification and the way to filter them, as this has been done for code smells [6]. For example, by studying the evolution of some AS, it would be possible to distinguish between the AS that exist by design (good code) from those that occurred by accident (bad code) [159]. It could be also interesting to analyse if a specific smell that tends to be present in the history of a project is also more critical with respect to the ones already removed. Moreover, the analysis of architectural smells correlations can be exploited to identify the architectural smells that tend to go together and that can be considered for this reason more critical with respect to the isolated smell.

- *Architectural smells correlations with code smells* Future works include the replication of the study considering different projects, as commercial or industrial projects, and the historical evolution of the projects. Moreover, other code smells other than these detected by the tools I adopted could have different relationships with architectural smells. Therefore, I believe there is the need of more empirical investigations in this domain, so as to understand (a) if the presence of other code smells implies the presence of one or more architectural smells, (b) if the independence between architectural smells and code smells is still true also by considering other architectural smells not currently detected by Arcan or by other available tools, (c) if the results obtained are valid for other projects, as the industrial ones. Moreover, as outlined by Kouroshfar et al [76] to improve the accuracy of bug prediction, one should also take the software architecture of the system into consideration. Hence I would like to study in the next future the possible correlations existing between architectural smells and bugs and also the possible correlations existing with different issues, e.g., issues detected through SonarQube.

- *Architectural Debt Index* – In the future works I would like to investigate the history index function on several projects and validate the ADI

in industrial projects. I would like to implements the ADI profiles functionality and exploit feedback of developers and practitioners on the usefulness of the index.

- *Architectural smells refactoring* – In future works I would like to apply the same methodology (see Chapter 8) on real world systems (industrial) and use professional developers to try to apply the refactoring steps or at least to evaluate the best choice both for the architectural smells to be solved first and the refactoring to apply. I aim to extend the experimentation by analyzing more projects through Arcan and remove the AS according to the insights described in the Chapter 8). Moreover, I aim to develop a recommending system of refactoring (Refactoring Advisor [14]) as a plugin of Arcan considering the work done by Caracciolo et al. [34] in Smalltalk. The plugin will be devote to the suggestion of the refactoring steps to remove the architectural smells, since their refactoring could be in many cases a very complex task. In this context, the aim is also to identify the most critical ones exploiting the Severity Index previously defined in order to provide some refactoring effort estimations and to define and implement other possible filters to remove false positive instances [6].

  In this context, I would like to work also on an automated refactoring scheduling approach, considering the possible conflicts and dependency relationships as described by Moghadam et al. [115] and their contribution in the improvement of specific metrics or quality indexes more focused on the evaluation of architectural issues.

BIBLIOGRAPHY

[1] Shameem Akhter and Jason Roberts. *Multi-core programming*. Vol. 33. Intel press Hillsboro, 2006.

[2] Hussain A. Al-Mutawa, Jens Dietrich, Stephen Marsland, and Catherine McCartin. "On the Shape of Circular Dependencies in Java Programs." In: *Proc. 23rd Australian Software Engineering Conference (ASWEC 2014)*. Sydney, Australia: IEEE, Apr. 2014, pp. 48–57. DOI: 10.1109/ASWEC.2014.15.

[3] A. Amirat, A. Bouchouk, M. O. Yeslem, and N. Gasmallah. "Refactor Software architecture using graph transformation approach." In: *Second International Conference on the Innovative Computing Technology (INTECH 2012)*. 2012, pp. 117–122. DOI: 10.1109/INTECH.2012.6457781.

[4] M. Aniche, G. Bavota, C. Treude, A. V. Deursen, and M. A. Gerosa. "A Validated Set of Smells in Model-View-Controller Architectures." In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 233–243. DOI: 10.1109/ICSME.2016.12.

[5] Francesca Arcelli Fontana, Vincenzo Ferme, and Stefano Spinelli. "Investigating the impact of code smells debt on quality code evaluation." In: *2012 Third International Workshop on Managing Technical Debt (MTD)*. 2012, pp. 15–22. DOI: 10.1109/MTD.2012.6225993.

[6] Francesca Arcelli Fontana, Vincenzo Ferme, and Marco Zanoni. "Filtering Code Smells Detection Results." In: *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. Poster track. Florence, Italy: IEEE, May 2015.

[7] Francesca Arcelli Fontana, Vincenzo Ferme, and Marco Zanoni. "Towards Assessing Software Architecture Quality by Exploiting Code Smell Relations." In: *Proceedings of the 2nd IEEE/ACM International Workshop on Software Architecture and Metrics (SAM 2015)*. Florence, Italy, May 2015, pp. 1–7. DOI: 10.1109/SAM.2015.8.

[8] Francesca Arcelli Fontana and Stefano Maggioni. "Metrics and Antipatterns for Software Quality Evaluation." In: *Proceedings of the 34th IEEE Software Engineering Workshop (SEW 2011)*. Limerick, Ireland: IEEE, June 2011, pp. 48–56. DOI: 10.1109/SEW.2011.13.

[9] Francesca Arcelli Fontana, Riccardo Roveda, and Marco Zanoni. "Discover knowledge on FLOSS projects through RepoFinder." In: *Proceedings of the International Conference on Knowledge Discovery and Information Retrieval (KDIR 2014)*. Rome, Italy: INSTICC Press, 2014, pp. 485–491. DOI: 10.5220/0005156704850491.

[10] Francesca Arcelli Fontana, Riccardo Roveda, and Marco Zanoni. "Technical Debt Indexes Provided by Tools: A Preliminary Discussion." In: *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*. 2016, pp. 28–31. DOI: 10.1109/MTD.2016.11.

[11] Francesca Arcelli Fontana, Riccardo Roveda, and Marco Zanoni. "Tool support for evaluating architectural debt of an existing system: An experience report." In: *Proceedings of the 31st ACM/SIGAPP Symposium on Applied Computing (SAC 2016)*. to appear. Pisa, Italy: ACM, Apr. 2016.

[12] Francesca Arcelli Fontana and Stefano Spinelli. "Impact of Refactoring on Quality Code Evaluation." In: *Proceedings of the 4th Workshop on Refactoring Tools*. WRT '11. Waikiki, Honolulu, USA: ACM, 2011, pp. 37–40. ISBN: 978-1-4503-0579-2. DOI: 10.1145/1984732.1984741. URL: http://doi.acm.org/10.1145/1984732.1984741.

[13] Francesca Arcelli Fontana and Stefano Spinelli. "Impact of refactoring on quality code evaluation." In: *Proc. 4th Workshop on Refactoring Tools (WRT '11)*. Waikiki, Honolulu, USA: ACM, 2011, pp. 37–40. ISBN: 978-1-4503-0579-2. DOI: 10.1145/1984732.1984741.

[14] Francesca Arcelli Fontana, Francesco Zanoni, and Marco Zanoni. "DCRA: a refactoring suggestion tool of Java code clones." In: *submitted to the 29th IEEE International Conference on Software Maintenance*. Submitted, waiting for review. IEEE, 2013, p. 10.

[15] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Riccardo Roveda. "Towards a Prioritization of Code Debt: A Code Smell Intensity Index." In: *Proc. Seventh Int'l Workshop on Managing Technical Debt (MTD 2015)*. In conjunction with ICSME 2015. Bremen, Germany: IEEE, Oct. 2015, pp. 16–24. DOI: 10.1109/MTD.2015.7332620.

[16] Francesca Arcelli Fontana, Riccardo Roveda, Marco Zanoni, Claudia Raibulet, and R. Capilla. "An Experience Report on Detecting and Repairing Software Architecture Erosion." In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 2016, pp. 21–30. DOI: 10.1109/WICSA.2016.37.

[17] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, and Marco Zanoni. "Automatic Detection of Instability Architectural Smells." In: *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*. ERA Track. Raleigh, North Carolina, USA: IEEE, Oct. 2016.

[18] Francesca Arcelli Fontana, Riccardo Roveda, Stefano Vittori, Andrea Metelli, Stefano Saldarini, and Francesco Mazzei. "On evaluating the impact of the refactoring of architectural problems on software quality." In: *Proceedings of the Scientific Workshop Proceedings of XP2016, Edinburgh, Scotland, UK, May 24, 2016*. 2016, p. 21. DOI: 10.1145/2962695.2962716. URL: http://doi.acm.org/10.1145/2962695.2962716.

[19] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Andrew Tamburri, Marco Zanoni, and Elisabetta Di Nitto. "Arcan: A Tool for Architectural Smells Detection." In: *Int'l Conf. Software Architecture (ICSA 2017) Workshops*. Gothenburg, Sweden, Apr. 2017, pp. 282–285. DOI: 10.1109/ICSAW.2017.16.

[20] Francesca Arcelli Fontana, Riccardo Roveda, Valentina Leonarduzzi, and Davide Taibi. "Are Architectural Smells Independent from Code Smells? An Empirical Study." In: *Submitted to The Journal of Information and Software Technology (IST)* (2018).

[21] Roberta Arcoverde, Alessandro Garcia, and Eduardo Figueiredo. "Understanding the longevity of code smells: preliminary results of an explanatory survey." In: *Fourth Workshop on Refactoring Tools 2011, WRT '11, Waikiki, Honolulu, HI, USA, May 22, 2011*. 2011, pp. 33–36. DOI: 10.1145/1984732.1984740.

[22] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. "Concurrency bugs in open source software: a case study." In: *J. Internet Services and Applications* 8.1 (2017), 4:1–4:15. DOI: 10.1186/s13174-017-0055-2.

[23] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0262024667.

[24] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[25] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. "An Experimental Investigation on the Innate Relationship Between Quality and Refactoring." In: *J. Syst. Softw.* 107.C (Sept. 2015), pp. 1–14. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.05.024. URL: http://dx.doi.org/10.1016/j.jss.2015.05.024.

[26] Yogesh Bhatia and Sanjeev Verma. "Deadlocks in distributed systems." In: *International Journal of Research* 1.9 (2014), pp. 1249–1252.

[27] David Binkley, Nicolas Gold, Mark Harman, Zheng Li, Kiarash Mahdavi, and Joachim Wegener. "Dependence Anti Patterns." In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*. L'Aquila, Italy: IEEE, Sept. 2008, pp. 25–34. ISBN: 978-1-4244-2776-5. DOI: 10.1109/ASEW.2008.4686318.

[28] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. "Does God Class Decomposition Affect Comprehensibility?" In: *IASTED Conf. on Software Engineering*. 2006.

[29] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004. ISBN: 020189551X.

[30] F. Bourquin and R. K. Keller. "High-impact Refactoring Based on Architecture Violations." In: *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. 2007, pp. 149–158. DOI: 10.1109/CSMR.2007.25.

[31] Kevin W. Bowyer, Nitesh V. Chawla, Lawrence O. Hall, and W. Philip Kegelmeyer. "SMOTE: Synthetic Minority Over-sampling Technique." In: *CoRR* abs/1106.1813 (2011).

[32] S. Brin and L. Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine." In: *Seventh International World-Wide Web Conference (WWW 1998)*. 1998. URL: http://ilpubs.stanford.edu:8090/361/.

[33] William Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley, 1998, p. 336. ISBN: 0-471-19713-0.

[34] Andrea Caracciolo, Bledar Aga, Mircea Lungu, and Oscar Nierstrasz. "Marea: A Semi-Automatic Decision Support System for Breaking Dependency Cycles." In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. 2016, pp. 482–492. DOI: 10.1109/SANER.2016.11. URL: http://dx.doi.org/10.1109/SANER.2016.11.

[35] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. "On the Impact of Refactoring Operations on Code Quality Metrics." In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 456–460. ISBN: 978-1-4799-6146-7. DOI: 10.1109/ICSME.2014.73. URL: http://dx.doi.org/10.1109/ICSME.2014.73.

[36] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. Vol. 10. MIT press, 2008.

[37] Alexander Chatzigeorgiou and Anastasios Manakos. "Investigating the Evolution of Bad Smells in Object-Oriented Code." In: *2010 Seventh Int'l Conf. the Quality of Information and Communications Tech.* IEEE, 2010, pp. 106–115. ISBN: 978-1-4244-8539-0. DOI: 10.1109/QUATIC.2010.16.

[38] Shyam R. Chidamber and Chris F. Kemerer. "A Metrics Suite for Object Oriented Design." In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. DOI: 10.1109/32.295895.

[39] Clarkware. *JDepend*. http://clarkware.com/software/JDepend.html. 2018.

[40] Peter Coad and Edward Yourdon. *Object-oriented Design*. Yourdon Press, 1991. ISBN: 0-13-630070-7.

[41] Corinna Cortes and Vladimir Vapnik. "Support-vector networks." In: *Machine Learning* 20.3 (1995), pp. 273–297. ISSN: 0885-6125. DOI: 10.1007/BF00994018.

[42] Ward Cunningham. "The WyCash portfolio management system." In: *OOPS Messenger* 4.2 (1993), pp. 29–30. DOI: 10.1145/157710.157715.

[43] Robert Dąbrowski, Krzysztof Stencel, and Grzegorz Timoszuk. "Software Is a Directed Multigraph." In: *Proc. 5th European Conf. Softw. Arch. (ECSA 2011)*. Essen, Germany: Springer, Sept. 2011, pp. 360–369. ISBN: 978-3-642-23798-0. DOI: 10.1007/978-3-642-23798-0_38.

[44] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. "An empirical investigation of an object-oriented design heuristic for maintainability." In: *Journal of Systems and Software* 65.2 (2003), pp. 127 –139. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(02)00054-7. URL: http://www.sciencedirect.com/science/article/pii/S0164121202000547.

[45] Jens Dietrich. "Upload Your Program, Share Your Model." In: *Proceedings of the 3rd Conference on Systems, Programming and Applications: Software for Humanity (SPLASH '12)*. Tucson, Arizona, USA: ACM, Oct. 2012, pp. 21–22. ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384727.

[46] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M. Ali Shah. "On the Existence of High-impact Refactoring Opportunities in Programs." In: *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*. ACSC '12. Melbourne, Australia: Australian Computer Society, Inc., 2012, pp. 37–48. ISBN: 978-1-921770-03-6. URL: http://dl.acm.org/citation.cfm?id=2483654.2483659.

[47] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. "Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt." In: *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Italy: ACM, 2015, pp. 50–60. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786848. URL: http://doi.acm.org/10.1145/2786805.2786848.

[48] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[49] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321127420. URL: http://portal.acm.org/citation.cfm?id=579257.

[50] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[51] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Yuanfang Cai. "Enhancing architectural recovery using concerns." In: *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 2011, pp. 552–555. DOI: 10.1109/ASE.2011.6100123.

[52] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. "Identifying Architectural Bad Smells." In: *CSMR 2009*. Germany: IEEE, 2009, pp. 255–258. DOI: 10.1109/CSMR.2009.59.

[53] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. "Toward a Catalogue of Architectural Bad Smells." In: *Proceedings of the 5th International Conference on the Quality of Software Architectures (QoSA 2009)*. East Stroudsburg, PA, USA: Springer Berlin Heidelberg, June 2009, pp. 146–162. ISBN: 978-3-642-02351-4. DOI: 10.1007/978-3-642-02351-4_10.

[54] Google, Inc. *CodePro Analytix User Guide*. https://developers.google.com/java-dev-tools/codepro/doc/. 2018.

[55] Ian Gorton and Liming Zhu. "Tool support for just-in-time architecture reconstruction and evaluation: an experience report." In: *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*. St. Louis, USA: IEEE, May 2005, pp. 514–523. DOI: 10.1109/ICSE.2005.1553597.

[56] Darryl Gove. *Multicore Application Programming: For Windows, Linux, and Oracle Solaris*. Addison-Wesley Professional, 2010.

[57] Jilles Van Gurp, Jan Bosch, and Sjaak Brinkkemper. "Design Erosion in Evolving Software Products." In: *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*. co-located with ICSM 2003. ESF RELEASE research network. Amsterdam, The Netherlands, Sept. 2003, pp. 134–139.

[58] Jilles van Gurp and Jan Bosch. "Design erosion: problems and causes." In: *Journal of Systems and Software* 61.2 (2002), pp. 105–119. DOI: 10.1016/S0164-1212(01)00152-2.

[59] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. "Some Code Smells Have a Significant but Small Effect on Faults." In: *ACM Trans. Softw. Eng. Methodol.* 23.4 (Sept. 2014), 33:1–33:39. ISSN: 1049-331X. DOI: 10.1145/2629648. URL: http://doi.acm.org/10.1145/2629648.

[60] Headway Software Technologies. *Structure101*. http://structure101.com/products/. 2018.

[61] Kennet Henningsson and Claes Wohlin. "Assuring fault classification agreement-an empirical evaluation." In: *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. IEEE. 2004, pp. 95–104.

[62] Myles Hollander and Douglas A. Wolfe. *Nonparametric Statistical Methods*. 2nd ed. New York: John Wiley & Sons, Aug. 1999.

[63] Sture Holm. "A Simple Sequentially Rejective Multiple Test Procedure." In: *Scandinavian Journal of Statistics* 6.2 (1979), pp. 65–70. ISSN: 03036898, 14679469.

[64] IBM Alphaworks. *SA4J — Structural Analysis for Java*. 2018.

[65] ISO. *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SquaRE) – System and software quality models*. site. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733. 2011.

[66] ISO/IEC. *ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. 2010.

[67] Intooitus. *inFusion*. http://www.intooitus.com/products/infusion. 2015.

[68] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F. Tichy. "Helgrind+: An efficient dynamic race detector." In: *IPDPS*. IEEE, 2009, pp. 1–13. DOI: 10.1109/IPDPS.2009.5160998.

[69] Deepal Jayasinghe and Pengcheng Xiong. "CORE: Visualization tool for fault localization in concurrent programs." In: *Internet: http://www.cc. gatech. edu/grads/i/ijayasin/resources/core_falcon. pdf* (2010).

[70] JetBrains s.r.o. *IntelliJ IDEA*. https://www.jetbrains.com/idea/. 2018.

[71] Cory J. Kapser and Michael W. Godfrey. ""Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software." In: *Empirical Softw. Engg.* 13.6 (Dec. 2008), pp. 645–692. ISSN: 1382-3256. DOI: 10.1007/s10664-008-9076-6. URL: http://dx.doi.org/10.1007/s10664-008-9076-6.

[72] R. Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, S. Haziyev, V. Fedak, and A. Shapochka. "A Case Study in Locating the Architectural Roots of Technical Debt." In: *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE 2015)*. Vol. 2. 2015, pp. 179–188. DOI: 10.1109/ICSE.2015.146.

[73] Joshua Kerievsky. "Refactoring to Patterns." In: *Extreme Programming and Agile Methods - XP/Agile Universe 2004, 4th Conference on Extreme Programming and Agile Methods, Calgary, Canada, August 15-18, 2004, Proceedings*. Ed. by Carmen Zannier, Hakan Erdogmus, and Lowell Lindstrom. Vol. 3134. Lecture Notes in Computer Science. Springer, 2004, p. 232. ISBN: 3-540-22839-X. DOI: 10.1007/978-3-540-27777-4_54. URL: http://dx.doi.org/10.1007/978-3-540-27777-4_54.

[74] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness." In: *Empirical Softw. Engg.* 17.3 (June 2012), pp. 243–275. ISSN: 1382-3256. DOI: 10.1007/s10664-011-9171-y. URL: http://dx.doi.org/10.1007/s10664-011-9171-y.

[75] M. V. Kosti, A. Ampatzoglou, A. Chatzigeorgiou, G. Pallas, I. Stamelos, and L. Angelis. "Technical Debt Principal Assessment Through Structural Metrics." In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2017, pp. 329–333. DOI: 10.1109/SEAA.2017.59.

[76] Ehsan Kouroshfar, Mehdi Mirakhorli, Hamid Bagheri, Lu Xiao, Sam Malek, and Yuanfang Cai. "A Study on the Role of Software Architecture in the Evolution and Quality of Software." In: *Proc. 12th Working Conf. Mining Software Repositories*. MSR '15. Florence, Italy: IEEE Press, 2015, pp. 246–257. ISBN: 978-0-7695-5594-2.

[77] Heiko Koziolek, Dominik Domis, Thomas Goldschmidt, and Philipp Vorst. "Measuring Architecture Sustainability." In: *IEEE Software* 30.6 (2013). DOI: 10.1109/MS.2013.101.

[78] G. Krasner and S. Pope. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System." In: *Journal of Object Oriented Programming* 1.3 (1988), pp. 26–49. URL: http://citeseer.ist.psu.edu/krasner88description.html.

[79] Klaus Krippendorff. *Content Analysis: An Introduction to Its Methodology (second edition)*. Sage Publications, 2004.

[80] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[81] Lattix, Inc. *Lattix Architect and Lattix Analyst*. http://lattix.com/lattix-architect-and-lattix-analyst. 2018.

[82] Jannik Laval and Stéphane Ducasse. "Resolving cyclic dependencies between packages with Enriched Dependency Structural Matrix." In: *Software: Practice and Experience* (Nov. 2012). URL: https://hal.inria.fr/hal-00748120.

[83] Duc Le and Nenad Medvidovic. "Architectural-based Speculative Analysis to Predict Bugs in a Software System." In: *Proc. 38th Int'l Conf. Software Eng. Companion*. ICSE '16. Austin, Texas: ACM, 2016, pp. 807–810. ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889260.

[84] Duc Le, Daniel Link, Yixue Zhao, Arman Shahbazian, Chris Mattmann, and Nenad Medvidovic. "Toward a Classification Framework for Software Architectural Smells." In: *Techincal Report csse.usc.edu* (2017). URL: http://csse.usc.edu/TECHRPTS/2017/TR_decay_arch.pdf.

[85] J.-L. Letouzey. "The SQALE method for evaluating Technical Debt." In: *MTD 2012*. 2012, pp. 31–36. DOI: 10.1109/MTD.2012.6225997.

[86] J. Letouzey and M. Ilkiewicz. "Managing Technical Debt with the SQALE Method." In: *IEEE Software* 29.6 (2012), pp. 44–51. ISSN: 0740-7459. DOI: 10.1109/MS.2012.129.

[87] Jean-Louis Letouzey. "The SQALE Method for Evaluating Technical Debt." In: *Proceedings of the Third International Workshop on Managing Technical Debt*. MTD '12. Zurich, Switzerland: IEEE Press, 2012, pp. 31–36. ISBN: 978-1-4673-1749-8. URL: http://dl.acm.org/citation.cfm?id=2666036.2666042.

[88] Wei Li and Raed Shatnawi. "An Empirical Study of the Bad Smells and Class Error Probability in the Post-release Object-oriented System Evolution." In: *J. Syst. Softw.* 80.7 (July 2007), pp. 1120–1128. ISSN: 0164-1212. DOI: 10.1016/j.jss.2006.10.018. URL: https://doi.org/10.1016/j.jss.2006.10.018.

[89] Zengyang Li, Paris Avgeriou, and Peng Liang. "A systematic mapping study on technical debt and its management." In: *Journal of Systems and Software* 101 (2015), pp. 193–220. ISSN: 0164-1212. DOI: 10.1016/j.jss.2014.12.027.

[90] Zengyang Li, Peng Liang, Paris Avgeriou, Nicolas Guelfi, and Apostolos Ampatzoglou. "An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt." In: *QoSA '14*. France: ACM, 2014, pp. 119–128. ISBN: 978-1-4503-2576-9. DOI: 10.1145/2602576.2602581.

[91] Andy Liaw and Matthew Wiener. "Classification and Regression by randomForest." In: *R News* 2.3 (2002), pp. 18–22. URL: http://CRAN.R-project.org/doc/Rnews/.

[92] Shiyao Lin, Andy Wellings, and Alan Burns. "Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages." In: *Concurrency and Computation: Practice and Experience* 25.16 (2013), pp. 2227–2251.

[93] Martin Lippert and Stephen Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, Apr. 2006, p. 286. ISBN: 978-0-470-85892-9.

[94] Hui Liu, Zhiyi Ma, Weizhong Shao, and Zhendong Niu. "Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort." In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 220–235. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.9.

[95] Angela Lozano and Michael Wermellinger. "Assessing the effect of clones on changeability." In: *IEEE International Conference on Software Maintenance*. ICSM '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 227–236. ISBN: 978-1-4244-2613-3.

[96] S. Lu, J. Tucek, F. Qin, and Y. Zhou. "AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants." In: *IEEE Micro* 27.1 (2007), pp. 26–35. ISSN: 0272-1732. DOI: 10.1109/MM.2007.5.

[97] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics." In: *ACM Sigplan Notices*. Vol. 43. 3. ACM. 2008, pp. 329–339.

[98] Isela Macía Bertrán. "On the Detection of Architecturally-Relevant Code Anomalies in Software Systems." PhD thesis. Rio de Janeiro: PUC-Rio, Departamento de Informática, 2013.

[99] Isela Macia Bertran, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt von Staa. "On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms." In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*. Szeged, Hungary, Mar. 2012. DOI: 10.1109/CSMR.2012.35.

[100] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa. "Supporting the identification of architecturally-relevant code anomalies." In: *Proc. of the 28th IEEE Intern.Conf. on Soft. Maint.(ICSM 2012)*. Italy: IEEE, 2012, pp. 662–665. DOI: 10.1109/ICSM.2012.6405348.

[101] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. "Are Automatically-detected Code Anomalies Relevant to Architectural Modularity?: An Exploratory Analysis of Evolving Systems." In: *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD '12)*. Potsdam, Germany: ACM, Mar. 2012, pp. 167–178. ISBN: 978-1-4503-1092-5. DOI: 10.1145/2162049.2162069.

[102] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt von Staa. "On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms." In: *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR 2012)*. Szeged, Hungary: IEEE Computer Society, Mar. 2012, pp. 277–286. DOI: 10.1109/CSMR.2012.35.

[103] Matthias Mair and Sebastian Herold. "Towards Extensive Software Architecture Erosion Repairs." In: *ECSA*. Ed. by Khalil Drira. Vol. 7957. Lecture Notes in Computer Science. Springer, 2013, pp. 299–306. ISBN: 978-3-642-39030-2. DOI: 10.1007/978-3-642-39031-9_25.

[104] Ruchika Malhotra and Megha Khanna. "An exploratory study for software change prediction in object-oriented systems using hybridized techniques." In: *Automated Software Eng.* (2016), pp. 1–45. ISSN: 1573-7535. DOI: 10.1007/s10515-016-0203-0.

[105] M. Mantyla, J. Vanhanen, and C. Lassenius. "A taxonomy and an initial empirical study of bad smells in code." In: *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. 2003, pp. 381–384. DOI: 10.1109/ICSM.2003.1235447.

[106] R. Marinescu, G. Ganea, and I. Verebi. "InCode: Continuous Quality Assessment and Improvement." In: *2010 14th European Conference on Software Maintenance and Reengineering*. 2010, pp. 274–275. DOI: 10.1109/CSMR.2010.44.

[107] Radu Marinescu. "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws." In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 350–359. ISBN: 0-7695-2213-0. URL: http://dl.acm.org/citation.cfm?id=1018431.1021443.

[108] Radu Marinescu. "Assessing technical debt by identifying design flaws in software systems." In: *IBM Journal of Research and Development* 56.5 (2012), 9:1–9:13. ISSN: 0018-8646. DOI: 10.1147/JRD.2012.2204512.

[109] Robert C. Martin. "Object Oriented Design Quality Metrics: An Analysis of dependencies." In: *ROAD* 2.3 (1995).

[110] Tom Mens and Arie Van Deursen. *Refactoring: Emerging Trends and Open Problems*. 2004. URL: \url{http://www.swen.uwaterloo.ca/~reface03/Papers/TomMens.pdf}.

[111] Tom Mens and Tom Tourwé. "A Survey of Software Refactoring." In: *IEEE Trans. Software Eng.* 30.2 (2004), pp. 126–139. DOI: 10.1109/TSE.2004.1265817. URL: http://dx.doi.org/10.1109/TSE.2004.1265817.

[112] *Metrics and Queries Documentation v.7.2*. hello2morrow GmbH. 2011.

[113] Bertrand Meyer. *Object-oriented Software Construction (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN: 0-13-629155-4.

[114] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells." In: *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015)*. Montreal, QC, Canada: IEEE, May 2015, pp. 51–60. DOI: 10.1109/WICSA.2015.12.

[115] Iman Hemati Moghadam and Mel Ó Cinnéide. "Resolving Conflict and Dependency in Refactoring to a Desired Design." In: *e-Informatica* 9.1 (2015), pp. 37–56. DOI: 10.5277/e-Inf150103. URL: http://dx.doi.org/10.5277/e-Inf150103.

[116] Naouel Moha and Yann-Gael Guéhéneuc. "Decor: a tool for the detection of design defects." In: *Proc. 22nd IEEE/ACM international conference on Automated software engineering (ASE '07)*. Atlanta, Georgia, USA: ACM, 2007, pp. 527–528. ISBN: 978-1-59593-882-4. DOI: 10.1145/1321631.1321727.

[117] Nachiappan Nagappan and Thomas Ball. "Using software dependencies and churn metrics to predict field failures: An empirical case study." In: *Proc. 1st Intern. Symp. on Empirical Software Eng. and Measurement (ESEM 2007)*. Madrid, Spain: IEEE, Sept. 2007, pp. 364–373. DOI: 10.1109/ESEM.2007.13.

[118] R.L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. "In Search of a Metric for Managing Architectural Technical Debt." In: *Proceedings of the 2012 Joint Working IEEE/IFIP Conf. on Soft. Arch. (WICSA) and European Conf. on Soft. Arch. (ECSA)*. Finland: IEEE, 2012, pp. 91–100. DOI: 10.1109/WICSA-ECSA.212.17.

[119] Mel Ó Cinnéide, Aiko Yamashita, and Steve Counsell. "Measuring Refactoring Benefits: A Survey of the Evidence." In: *Proceedings of the 1st International Workshop on Software Refactoring*. IWoR 2016. Singapore, Singapore: ACM, 2016, pp. 9–12. ISBN: 978-1-4503-4509-5. DOI: 10.1145/2975945.2975948. URL: http://doi.acm.org/10.1145/2975945.2975948.

[120] Odysseus Software GmbH. *STAN*. http://stan4j.com/. 2018.

[121] W. Oizumi, A. Garcia, M. Ferreira, A. von Staa, and T.E. Colanzi. "When Code-Anomaly Agglomerations Represent Architectural Problems? An Exploratory Study." In: *Software Engineering (SBES), 2014 Brazilian Symposium on*. 2014, pp. 91–100. DOI: 10.1109/SBES.2014.18.

[122] Willian Nalepa Oizumi, Alessandro F. Garcia, Leonardo da Silva Sousa, Bruno Barbieri Pontes Cafeo, and Yixue Zhao. "Code anomalies flock together: exploring code anomaly agglomerations for locating design problems." In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, pp. 440–451. DOI: 10.1145/2884781.2884868. URL: http://doi.acm.org/10.1145/2884781.2884868.

[123] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. "The evolution and impact of code smells: A case study of two open source systems." In: *2009 3rd Int'l Symp. Empirical Software Eng. and Measurement*. 2009, pp. 390–400. DOI: 10.1109/ESEM.2009.5314231.

[124] S.M. Olbrich, D.S. Cruzes, and Dag I K Sjoberg. "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems." In: *IEEE International Conference on Software Maintenance (ICSM 2010)*. 2010, p. 10. DOI: 10.1109/ICSM.2010.5609564.

[125] Ali Ouni, Marouane Kessentini, and Houari Sahraoui. "Chapter Four - Multiobjective Optimization for Software Refactoring and Evolution." In: *Advances in Computers*. Ed. by Ali Hurson. Vol. 94. Elsevier, 2014, pp. 103 –167. DOI: http://dx.doi.org/10.1016/B978-0-12-800161-5.00004-9.

[126] Tosin Daniel Oyetoyan, Jean-Rémy Falleri, Jens Dietrich, and Kamil Jezek. "Circular dependencies and change-proneness: An empirical study." In: *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. 2015, pp. 241–250. DOI: 10.1109/SANER.2015.7081834. URL: http://dx.doi.org/10.1109/SANER.2015.7081834.

[127] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. "Mining Version Histories for Detecting Code Smells." In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 462–489. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2372760.

[128] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto. "Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells." In: *2016 IEEE International Conference Software Maintenance and Evolution (ICSME)*. 2016, pp. 244–255. DOI: 10.1109/ICSME.2016.27.

[129] Sangmin Park, Richard W Vuduc, and Mary Jean Harrold. "Falcon: fault localization in concurrent programs." In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM. 2010, pp. 245–254.

[130] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the Study of Software Architecture." In: *SIGSOFT Softw. Eng. Notes* 17.4 (Oct. 1992), pp. 40–52. ISSN: 0163-5948. DOI: 10.1145/141874.141884. URL: http://doi.acm.org/10.1145/141874.141884.

[131] Ralph Peters and Andy Zaidman. "Evaluating the Lifespan of Code Smells using Software Repository Mining." In: *2012 16th European Conf. Softw. Maintenance and ReEng.* IEEE, 2012, pp. 411–416. ISBN: 978-0-7695-4666-7. DOI: 10.1109/CSMR.2012.79.

[132] Błażej Pietrzak and Bartosz Walter. "Leveraging Code Smell Detection with Inter-smell Relations." In: *Extreme Programming and Agile Processes in Software Engineering*. Ed. by Pekka Abrahamsson, Michele Marchesi, and Giancarlo Succi. Vol. 4044. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 75–84. ISBN: 978-3-540-35094-1. DOI: 10.1007/11774129_8.

[133] Aoun Raza, Gunther Vogel, and Erhard Plödereder. "Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering." In: *Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe 2006)*. Porto, Portugal: Springer Berlin Heidelberg, June 2006, pp. 71–82. ISBN: 978-3-540-34664-7. DOI: 10.1007/11767077_6.

[134] D. Romano, P. Raila, M. Pinzger, and F. Khomh. "Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes." In: *Proc. 19th Working Conf. on Reverse Eng. (WCRE 2012)*. Kingston, Ontario, Canada: IEEE, Oct. 2012, pp. 437–446. DOI: 10.1109/WCRE.2012.53.

[135] Riccardo Roveda, Francesca Arcelli Fontana, and Damian Andrew Tamburri. "Automated detection and Evaluation of Architecture Smells: a Mixed-Methods Study." In: *to be Submitted* (2018).

[136] Riccardo Roveda, Francesca Arcelli Fontana, Claudia Raibulet, Marco Zanoni, and Federico Rampazzo. "Does the Migration to GitHub Relate to Internal Software Quality?" In: *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering*. 2017, pp. 293–300. ISBN: 978-989-758-250-9. DOI: 10.5220/0006367402930300.

[137] Riccardo Roveda, Francesca Arcelli Fontana, Ilaria Pigazzini, Marco Zanoni, and Paris Avgeriou. "A Study on Architectural Smells Prediction and Evolution." In: *Submitted to Research Track of 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018)*. IEEE, 2018.

[138] Per Runeson and Martin Höst. "Guidelines for Conducting and Reporting Case Study Research in Software Engineering." In: *Empirical Softw. Engg.* 14.2 (Apr. 2009), pp. 131–164. ISSN: 1382-3256. DOI: 10.1007/s10664-008-9102-8. URL: http://dx.doi.org/10.1007/s10664-008-9102-8.

[139] Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. "Refactoring for Software Architecture Smells." In: *Proceedings of the 1st International Workshop on Software Refactoring*. IWoR 2016. Singapore, Singapore: ACM, 2016, pp. 1–4. ISBN: 978-1-4503-4509-5. DOI: 10.1145/2975945.2975946. URL: http://doi.acm.org/10.1145/2975945.2975946.

[140] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. "Building Empirical Support for Automated Code Smell Detection." In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '10. Bolzano-Bozen, Italy: ACM, 2010, 8:1–8:10. ISBN: 978-1-4503-0039-1. DOI: 10.1145/1852786.1852797. URL: http://doi.acm.org/10.1145/1852786.1852797.

[141] Scientific Toolworks, Inc. *Understand Your Code*. https://scitools.com/. 2018.

[142] Robert Sedgewick and Kevin Wayne. *Algorithms 4thEd*. Addison-Wesley, 2011. ISBN: 978-0-321-57351-3.

[143] S. M. A. Shah, J. Dietrich, and C. McCartin. "Making Smart Moves to Untangle Programs." In: *2012 16th European Conference on Software Maintenance and Reengineering*. 2012, pp. 359–364. DOI: 10.1109/CSMR.2012.44.

[144] Lakshitha de Silva and Dharini Balasubramaniam. "Controlling software architecture erosion: A survey." In: *Journal of Systems and Software* 85.1 (2012). DOI: 10.1016/j.jss.2011.07.036.

[145] Dag I. K. Sjoberg, Aiko Yamashita, Bente Anda, Audris Mockus, and Tore Dyba. "Quantifying the Effect of Code Smells on Maintenance Effort." In: *IEEE Trans. Softw. Eng.* 39.8 (2013), pp. 1144–1156. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.89. URL: http://dx.doi.org/10.1109/TSE.2012.89.

[146] SonarSource S.A. *SonarQube*. http://www.sonarqube.org/. 2018.

[147] Stal, Michael. *Software architecture refactoring.* Software Architecture Refactoring, Siemens AG Corporate Technology. In Tutorial, in The International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA). http://stal.blogspot.in/2007/01/architecture-refactoring.html. 2007.

[148] William Stallings. *Operating Systems: Internals and Design Principles*. Pearson, 2012.

[149] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN: 0128013974, 9780128013977.

[150] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan. "Predicting Bugs Using Antipatterns." In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 270–279. DOI: 10.1109/ICSM.2013.38.

[151] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E. Hassan, and Meiyappan Nagappan. "Predicting Bugs Using Antipatterns." In: *Proceedings of the 2013 IEEE International Conference on Software Maintenance*. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 270–279. ISBN: 978-0-7695-4981-1. DOI: 10.1109/ICSM.2013.38. URL: http://dx.doi.org/10.1109/ICSM.2013.38.

[152] Davide Taibi. *Raw Data: Are Architectural Smells independent from Code Smells? An empirical study*. https://data.mendeley.com/datasets/tnhk383zvz/. 2017. DOI: http://dx.doi.org/10.17632/tnhk383zvz.2.

[153] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. "How developers perceive smells in source code: A replicated study." In: *Information and Software Technology* 92.Supplement C (2017), pp. 223 –235. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.08.008. URL: http://www.sciencedirect.com/science/article/pii/S0950584916304128.

[154] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies." In: *Proc. 17th Asia Pacific Software Engineering Conference (APSEC 2010)*. Sydney, Australia: IEEE, 2010, pp. 336–345. DOI: 10.1109/APSEC.2010.46.

[155] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. "Recommending Refactorings to Reverse Software Architecture Erosion." In: *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*. CSMR '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 335–340. ISBN: 978-0-7695-4666-7. DOI: 10.1109/CSMR.2012.40. URL: http://dx.doi.org/10.1109/CSMR.2012.40.

[156] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. "Qualitas.class Corpus: A Compiled Version of the Qualitas Corpus." In: *Software Engineering Notes* 38.5 (2013), pp. 1–4.

[157] Edith Tom, Aybüke Aurum, and Richard T. Vidgen. "An exploration of technical debt." In: *Journal of Systems and Software* 86.6 (2013), pp. 1498–1516. DOI: 10.1016/j.jss.2012.12.052.

[158] N. L. Tran, S. Skhiri, A. Lesuisse, and E. Zimányi. "AROM: Processing big data with Data Flow Graphs and functional programming." In: *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. 2012, pp. 875–882. DOI: 10.1109/CloudCom.2012.6427487.

[159] S. Vaucher, F. Khomh, N. Moha, and Y. G. Gueheneuc. "Tracking Design Smells: Lessons from a Study of God Classes." In: *2009 16th Working Conf. Reverse Eng.* 2009, pp. 145–154. DOI: 10.1109/WCRE.2009.23.

[160] William C. Wake. *Refactoring Workbook*. 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321109295.

[161] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN: 0-7923-8682-5.

[162] Lu Xiao, Yuanfang Cai, and Rick Kazman. "Titan: A Toolset That Connects Software Architecture with Quality Analysis." In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Hong Kong, China: ACM, Nov. 2014, pp. 763–766. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2661677.

[163] Aiko Yamashita, Marco Zanoni, Francesca Arcelli Fontana, and Bartosz Walter. "Inter-smell relations in industrial and open source systems: A replication and comparative analysis." In: *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME2015)*. Bremen, Germany: IEEE, Sept. 2015, pp. 121–130. DOI: 10.1109/ICSM.2015.7332458.

[164] Robert K. Yin. *Case Study Research: Design and Methods, 4th Edition (Applied Social Research Methods, Vol. 5)*. 4th. SAGE Publications, Inc, 2009. ISBN: 9781412960991.

[165] Norihiro Yoshida, Tsubasa Saika, Eunjong Choi, Ali Ouni, and Katsuro Inoue. "Revisiting the relationship between code smells and refactoring." In: *24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016*. 2016, pp. 1–4. DOI: 10.1109/ICPC.2016.7503738. URL: http://dx.doi.org/10.1109/ICPC.2016.7503738.

[166] Noriaki Yoshiura and Wei Wei. "Static data race detection for java programs with dynamic class loading." In: *International Conference on Internet and Distributed Computing Systems*. Cham: Springer International Publishing, 2014, pp. 161–173. ISBN: 978-3-319-11692-1. DOI: 10.1007/978-3-319-11692-1_14.

[167] O. Zimmermann. "Architectural Refactoring: A Task-Centric View on Software Evolution." In: *IEEE Software* 32.2 (2015), pp. 26–29. ISSN: 0740-7459. DOI: doi.ieeecomputersociety.org/10.1109/MS.2015.37.

[168] hello2morrow. *Sonargraph*. https://www.hello2morrow.com/products/sonargraph. 2018.

APPENDIX

A.1 THE ANALYZED PROJECTS

The Table A.1 is used in Section 6.2.

Table A.1: Number of Architectural Smells, Group of Code smells and Code smells infecting the analyzed projects

Project	Architectural Smells				Groups of Code Smells				Code Smells														
	UD	HL	CD	MAS	Bloat.	Disp.	Enc.	OOA	AS	BCSA	DsP	CC	DC	LC	LzC	LM	LPL	MFnC	MC	RPB	SC	SG
aoi	6	7	11110	11123	225	191	42	31	31	0	42	78	188	0	3	108	39	0	0	0	3	0
argouml	3	25	1833	1861	854	1344	62	17	17	3	61	333	1294	0	50	429	89	3	1	0	6	2
aspectj	8	52	48064	48124	974	1564	353	93	93	5	353	437	1501	0	63	352	183	2	0	6	10	2
axion	1	5	158	164	85	40	8	3	3	0	8	31	39	0	1	46	7	1	0	0	0	0
azureus	6	168	162357	162531	1193	704	173	139	139	65	111	428	494	0	210	462	303	0	62	139	19	25
cjdbc	7	35	1632	1674	365	136	21	54	54	0	21	140	132	0	4	165	60	0	0	0	6	0
castor	9	58	1510	1577	1511	1017	32	11	11	4	32	511	956	0	61	573	427	0	0	4	4	1
cayenne	2	1	12	15	1241	552	18	25	25	4	18	479	343	0	209	655	106	1	0	32	0	25
checkstyle	4	3	253	260	446	50	10	5	5	2	10	186	50	0	0	158	100	2	0	2	0	0
cobertura	6	4	38	48	78	49	26	3	3	2	26	30	48	0	1	31	17	0	0	0	0	0
collections	4	5	360	369	241	411	2	2	2	1	2	77	409	0	2	107	57	0	0	0	0	4
colt	6	9	885	900	85	109	6	22	22	1	6	17	95	0	14	50	18	0	0	2	5	0
columba	5	60	2294	2359	623	146	24	29	29	0	24	169	134	0	12	253	201	0	0	0	0	0
compiere	9	19	8282	8310	1351	1292	81	734	734	2	81	412	1265	2	27	516	415	6	0	1	5	1
derby	4	51	9648	9703	1455	739	163	98	98	11	163	542	581	0	158	604	303	6	0	9	16	6
displaytag	1	5	117	123	197	81	0	2	2	2	0	61	81	0	0	79	57	0	0	1	0	0
drawswf	2	14	364	380	102	57	7	12	12	10	7	40	55	0	2	53	9	0	0	10	0	1
emma	3	8	187	198	79	41	19	0	0	0	19	21	39	0	2	28	30	0	0	0	0	3
exoportal	0	49	370	419	998	269	44	77	77	11	44	263	228	0	41	357	377	1	0	8	7	3
findbugs	10	13	9111	9134	449	97	14	34	34	0	14	174	72	0	25	202	72	1	0	0	6	0
fitjava	2	0	32	34	32	77	31	15	15	1	31	9	75	0	2	15	8	0	0	0	0	1
fitlibraryforfitnesse	6	38	2436	2480	607	161	75	27	27	11	75	152	102	0	59	229	224	2	0	4	3	0

Continued on next page

Table A.1 – continued from previous page

Project	Architectural Smells				Groups of Code Smells				Code Smells														
	UD	HL	CD	MAS	Bloat.	Disp.	Enc.	OOA	ASG	BCSA	DsP	CC	DC	LC	LzC	LM	LPL	MFnC	MC	RPB	SC	SG
freecol	15	20	41088	41123	323	86	6	6	6	0	6	123	81	0	5	159	41	0	0	0	0	0
freecs	4	6	742	752	78	66	26	13	13	0	26	27	66	0	0	29	21	1	0	0	0	0
freemind	9	16	4350	4375	181	54	48	27	27	10	11	64	43	0	11	89	28	0	37	34	6	3
galleon	8	6	1788	1802	153	125	18	20	20	0	18	66	125	0	0	50	37	0	0	0	3	0
ganttproject	5	20	1852	1877	222	51	16	13	13	0	16	79	39	0	12	103	39	1	0	0	3	0
hadoop	9	48	6865	6922	994	789	46	103	103	11	46	326	753	0	36	550	117	1	0	10	11	6
heritrix	6	16	996	1018	261	63	20	37	37	0	20	89	61	0	2	138	34	0	0	0	1	0
hsqldb	9	11	10606	10626	257	193	44	42	42	3	44	112	162	1	31	109	35	0	0	1	8	0
htmlunit	2	7	10408	10417	357	118	0	0	0	0	0	176	117	0	1	178	3	0	0	0	0	0
informa	3	3	49	55	65	42	0	8	8	4	0	24	42	0	0	31	10	0	0	3	0	0
ireport	9	44	14344	14397	893	838	36	39	39	1	36	303	829	2	9	270	318	0	0	1	8	0
itext	6	6	3519	3531	215	89	18	17	17	0	18	113	63	1	26	73	28	0	0	0	3	0
ivatagroupware	3	20	14	37	105	25	2	1	1	0	2	38	19	0	6	44	23	0	0	0	0	0
jag	1	7	422	430	72	29	13	11	11	0	13	19	20	0	9	30	23	0	0	0	4	0
james	3	6	100	109	144	67	2	1	1	0	2	56	60	0	7	81	7	0	0	1	0	1
jasml	0	0	0	0	21	5	27	3	3	0	27	10	3	0	2	4	5	2	0	0	0	0
jasperreports	7	21	1829	1857	820	306	0	3	3	1	0	285	300	0	6	282	253	0	0	0	1	0
javacc	4	2	47	53	70	57	38	23	23	2	38	30	54	0	3	28	11	1	0	0	9	0
jboss	9	93	2658	2760	3364	1501	305	358	358	17	305	914	1330	0	171	1251	1197	2	0	11	40	3
jchempaint	5	37	1737	1779	914	823	38	44	44	4	38	386	693	0	130	426	102	0	0	1	8	0
jedit	8	14	11509	11531	257	138	47	22	22	1	47	104	89	0	49	131	20	2	0	1	0	0
jena	6	21	6257	6284	324	117	71	115	115	28	66	134	73	0	44	153	36	1	5	71	13	7
jext	6	13	1255	1274	277	153	27	16	16	1	27	124	150	0	3	123	29	1	0	1	2	0

Continued on next page

Table A.1 – continued from previous page

Project	Architectural Smells				Groups of Code Smells				Code Smells														
	UD	HL	CD	MAS	Bloat.	Disp.	Enc.	OOA	ASG	BCSA	DsP	CC	DC	LC	LzC	LM	LPL	MFnC	MC	RPB	SC	SG
jFin DateMath	0	2	0	2	48	20	3	3	3	0	3	19	18	0	2	25	4	0	0	0	0	0
jfreechart	9	15	245	269	440	331	5	29	29	0	5	134	326	0	5	214	92	0	0	0	1	0
jgraph	6	8	908	922	81	67	37	44	44	0	37	26	56	0	11	44	11	0	0	0	0	0
jgraphpad	4	4	186	194	166	35	21	32	32	0	21	56	35	0	0	64	46	0	0	1	3	1
jgrapht	1	6	58	65	96	19	8	2	2	0	8	33	16	0	3	58	5	0	0	0	0	0
jgroups	4	7	859	870	301	144	13	24	24	1	13	113	135	1	9	169	18	0	0	1	1	0
jhotdraw	8	22	827	857	356	274	9	8	8	4	9	140	241	0	33	146	70	0	0	0	0	0
jmeter	7	35	4464	4506	498	222	1	0	0	0	1	117	170	0	52	196	185	0	0	0	0	0
jmoney	0	3	302	305	34	14	0	4	4	1	0	10	7	0	7	14	10	0	0	0	0	0
joggplayer	3	3	212	218	122	18	13	16	16	1	13	41	16	0	2	44	37	0	0	0	5	0
jparse	1	1	122	124	41	21	7	3	3	1	1	14	21	0	0	15	12	0	6	0	1	0
jpf	1	2	42	45	35	8	0	4	4	0	0	11	7	0	1	19	5	0	0	0	0	0
jrefactory	4	37	2901	2942	805	309	33	23	23	29	27	260	309	0	0	310	233	2	6	74	6	5
jruby	12	46	147592	147650	636	208	68	45	45	9	68	304	179	0	29	272	58	2	0	4	9	6
jspwiki	6	17	1610	1633	317	61	9	26	26	0	9	80	60	0	1	139	98	0	0	0	0	0
jsXe	6	7	368	381	72	29	6	2	2	0	6	24	14	0	15	34	12	2	0	0	0	1
jtopen	1	3	3690	3694	982	748	10	50	50	0	10	461	655	0	93	311	209	1	0	0	6	0
jung	2	14	200	216	173	112	6	10	10	0	6	48	112	0	0	92	33	0	0	0	2	0
junit	2	11	183	196	79	49	3	3	3	7	3	15	46	0	3	35	29	0	0	17	1	6
log4j	3	13	411	427	190	91	3	11	11	2	3	61	89	0	2	107	22	0	0	0	2	1
lucene	2	66	3786	3854	608	364	44	26	26	3	44	190	353	0	11	284	134	0	0	0	2	4
marauroa	5	12	264	281	94	30	6	0	0	0	6	26	28	0	2	57	11	0	0	0	0	0
maven	6	27	691	724	331	99	1	0	0	5	1	115	80	0	19	149	67	0	0	5	0	1

Continued on next page

Table A.1 – continued from previous page

Project	Architectural Smells				Groups of Code Smells				Code Smells														
	UD	HL	CD	MAS	Bloat.	Disp.	Enc.	OOA	ASG	BCSA	DsP	CC	DC	LC	LzC	LM	LPL	MFnC	MC	RPB	SC	SG
megamek	4	20	11532	11556	581	1076	61	24	24	1	61	387	1076	0	0	182	7	5	0	1	6	0
mvnforum	3	26	1071	1100	331	294	10	18	18	1	10	118	240	0	54	128	85	0	0	1	2	0
myfaces core	7	41	1267	1315	924	1849	5	5	5	16	5	294	1830	0	19	299	331	0	0	53	0	35
nakedobjects	2	99	2578	2679	1082	322	44	34	34	0	44	398	223	0	99	557	127	0	0	0	2	1
nekohtml	1	2	15	18	13	15	0	1	1	0	0	6	6	0	9	7	0	0	0	0	0	0
openjms	2	19	301	322	230	60	0	3	3	0	0	64	46	0	14	127	39	0	0	0	0	0
oscache	1	4	45	50	63	8	5	9	9	0	5	16	8	0	0	25	22	0	0	0	1	0
picocontainer	1	4	129	134	53	11	0	0	0	1	0	21	7	0	4	32	0	0	0	18	0	1
pmd	2	17	310	329	379	77	18	5	5	4	18	120	52	0	25	167	92	0	0	2	0	2
poi	13	43	3491	3547	956	306	25	20	20	7	25	408	270	0	36	432	113	3	0	0	1	2
pooka	7	8	10070	10085	113	93	64	48	48	22	53	29	75	0	18	74	10	0	11	12	5	1
proguard	2	15	287	304	278	34	73	1	1	0	73	118	34	0	0	124	35	1	0	0	1	0
quartz	1	8	198	207	93	48	0	0	0	0	0	31	44	0	4	45	17	0	0	0	0	0
quickserver	1	6	288	295	89	57	13	9	9	0	13	29	57	0	0	38	22	0	0	0	1	0
quilt	0	3	70	73	48	10	7	6	6	0	7	8	10	0	0	23	17	0	0	0	0	0
roller	6	23	371	400	402	190	45	54	54	0	45	96	189	0	1	165	141	0	0	0	5	0
rssowl	10	51	17009	17070	254	192	37	39	39	0	37	95	111	0	81	110	29	20	0	0	2	0
sablecc	1	1	44	46	92	80	11	4	4	0	11	25	49	0	31	30	37	0	0	0	3	0
sandmark	2	23	760	785	403	185	76	37	37	14	76	149	168	0	17	185	69	0	0	24	2	2
springframework	7	100	3604	3711	2075	815	36	27	27	2	36	614	719	0	96	777	683	1	0	1	0	0
squirrelsql	0	2	231	233	19	1	0	0	0	0	0	6	0	0	1	12	1	0	0	0	0	0
struts	5	43	1241	1289	790	431	30	30	30	4	30	331	397	0	34	433	26	0	0	3	7	1
sunflow	3	7	850	860	94	33	4	0	0	0	4	35	33	0	0	37	22	0	0	0	0	0

Continued on next page

Table A.1 – continued from previous page

Project	Architectural Smells				Groups of Code Smells				Code Smells														
	UD	HL	CD	MAS	Bloat.	Disp.	Enc.	OOA	ASG	BCSA	DsP	CC	DC	LC	LzC	LM	LPL	MFnC	MC	RPB	SC	SG
tapestry	7	29	971	1007	745	49	5	5	5	0	5	245	49	0	0	395	105	0	0	0	0	0
tomcat	5	35	1891	1931	612	909	22	68	68	3	22	229	795	0	114	263	119	1	0	3	10	1
trove	0	0	17	17	22	8	0	0	0	0	0	8	5	0	3	9	5	0	0	0	0	0
velocity	2	14	287	303	172	53	11	4	4	0	11	64	52	0	1	92	15	1	0	0	0	0
wct	4	35	685	724	259	162	8	7	7	0	8	99	126	0	36	141	19	0	0	0	0	0
webmail	4	7	117	128	50	29	2	4	4	1	2	17	27	0	2	29	4	0	0	2	0	0
weka	4	32	5179	5215	654	639	56	85	85	0	56	206	578	0	61	334	114	0	0	0	4	0
xalan	4	21	8027	8052	567	439	33	7	7	10	33	223	313	0	126	228	115	1	0	8	1	5
xerces	2	12	1130	1144	367	439	31	13	13	5	31	116	409	0	30	104	146	1	0	0	2	0
xmojo	0	1	4	5	10	16	3	2	2	0	3	3	14	0	2	6	1	0	0	0	1	0
antlr	4	8	601	613	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
ant	5	13	2511	2529	551	152	11	3	3	24	8	135	102	0	50	213	203	0	3	74	1	3