TABLE II: Kinds of Agglomerations Supported by JSpIRIT

Agglomeration	Short description
Intra-component	This agglomeration identifies smells that are implemented by the same architectural component
Cross-component	This agglomeration identifies smells that are syntactically related but that they are implemented by different architectural components
Hierarchical	This agglomeration identifies smells that occur across the same inheritance tree involving one or more components

TABLE I: Kinds of Smells Supported by JSpIRIT

Code smell	Short description
Brain Class	Complex class that accumulates intelligence by brain methods
Brain Method	Long and complex method that centralizes the intelligence of a class
Data Class	Class that contains data but not behavior related to the data
Dispersed Coupling	Method that calls more methods of single external class that their own
Feature Envy	Method that calls one or few methods of several classes
God Class	Long and complex class that centralizes the intelligence of the system
Intensive Coupling	Method that calls several methods that are implemented in one or few classes
Refused Parent Bequest	Subclass that does not use the protected methods of its superclass
Shotgun Surgery	Method called by many methods that are implemented in different classes
Tradition Breaker	Subclass that does not specialize the superclass

Smell 1: God Class

Kind of like a Blob
See above image

Remedy: Extract Class

Smell 2: Brain Method

See above image

Remedy: Extract Method

Smell 3: Brain Class

See above image

Smell 4: Data Class

See above image

Remedies: Move Method, Extract Method

Smell 5: Dispersed Coupling

Smell 6: Feature Envy

This smell occurs when “a method is more interested in another class than the one it is actually in”. Thus, a method affected by Feature Envy is not correctly placed, since it exhibits high coupling with a class different than the one where it is located.

Remedy: Move Method

Smell 7: Intensive Coupling

Smell 8: Refused Parent Bequest

This rather potent odor results when subclasses inherit code that they don’t want. In some cases, a subclass may “refuse the bequest” by providing a do-nothing implementation of an inherited method.

Remedies: Push Down Field, Push Down Method

See above image

Smell 9: Shotgun Surgery

Shotgun Surgery refers to when a single change is made to multiple classes simultaneously.

Remedies: Move Method, Move Field

See above image

Smell 10: Tradition Breaker

See above image

The *Blob* (also called God class [14]) corresponds to a large controller class that ***depends on data stored*** in surrounding data classes. A large class declares ***many*** fields and methods with a ***low*** cohesion. A controller class monopolises most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes [44]. Controller classes can be identified using suspicious names such as `Process`, `Control`, `Manage`, `System`, and so on. A data class contains only data and performs no processing on these data. It is composed of highly cohesive fields and accessors.
The *Functional Decomposition* antipattern may occur if experienced procedural developers with little knowledge of object-orientation implement an object-oriented system. Brown describes this antipattern as "a 'main' routine that calls numerous subroutines". The Functional Decomposition design smell consists of a main class, *i.e.*, a class with a procedural name, such as `Compute` or `Display`, in which inheritance and polymorphism are scarcely used, that is ***associated with small classes***, which declare ***many*** private fields and implement only a ***few*** methods.
The *Spaghetti Code* is an antipattern that is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring ***long methods*** with ***no parameters***, and utilising ***global variables***. ***Names*** of classes and methods may suggest ***procedural*** programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms, ***polymorphism*** and ***inheritance***.
The *Swiss Army Knife* refers to a tool fulfilling a wide range of needs. The Swiss Army Knife design smell is a complex class that offers a ***high*** number of services, for example, a complex class implementing a ***high*** number of interfaces. A Swiss Army Knife is different from a Blob, because it exposes a ***high*** complexity to address all foreseeable needs of a part of a system, whereas the Blob is a singleton monopolising all processing and data of a system. Thus, several Swiss Army Knives may exist in a system, for example utility classes.

The key concepts are in bold and italics.

Smell 11: Blob (Design Smell)

These classes are usually characterized by a huge size, a large number of attributes and methods and a high number of dependencies with data classes. This smell involves low cohesive classes that are responsible for the management of different functionalities.

Remedy: Extract Class

See above image

Smell 12: Swiss Army Knife (Design Smell)

See above image

Smell 13: Functional Decomposition (Design Smell)

See above image

Smell 14: Spaghetti Code (Design Smell)

See above image

Smell 15: State Checking

Could not find anything. LOL

Smell 16: Long Method

This smell arises when a method implements a main functionality, together with auxiliary functions that should be managed in different methods. It is related to how many responsibilities a method manages, i.e. whether a method violates the single responsibility principle.

Remedy: Extract Method

Smell 17: Promiscuous Package

A package can be considered as promiscuous if it contains classes implementing too many features, making it too hard to understand and maintain. As for Blob, this smell arises when the package has low cohesion, since it manages different responsibilities.

Remedy: Extract Package

Smell 18: Misplaced Class

A Misplaced Class smell suggests a class that is in a package that contains other classes not related to it.

Remedy: Move Class

Smell 19: Message Chain

A message chain occurs when a client requests another object, that object requests yet another one, and so on. These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.

Remedies: Extract Method, Move Method

Smell 20: Duplicated Code

Remedy: Extract Method

Smell 21: Comments

Comments represent a failure to express an idea in the code. Try to make your code self-documenting or intention-revealing.

Remedies: Extract Method, Rename Method, Introduce Assertion

Smell 22: Large Class

The presence of too many instance variables usually indicates that a class is trying to do too much. In general, large classes typically contain too many responsibilities.

Remedy: Extract Class

Smell 23: Divergent Change

Divergent Change smells **occur when a single class needs to be edited many times when changes are made outside the class**. The consequence of this is that you can easily cause errors whenever a change is made that the class does not expect, limiting how quickly you can make changes to the code.

Remedy: Extract Class

Smell 24: Long Parameter List

Methods that take too many parameters produce client code that is awkward and difficult to work with.

Remedies: Introduce Parameter Object, Replace Parameter with Method, Preserve Whole Object