Name	*Large Class*

Description

Large Class is described by (FOWLER, 2004) as a class that tries to do too much. On object oriented models, a class should represent a single abstraction of a given domain. When a class becomes too large, it is often an indication that this OO good practice is not respected. Then a *Large Class* is often traduced as a class with too many attributes or too many methods.

Source code detection

Large Class commonly use one of the following information to identify the code smell:

- The class number of attributes and methods (NOAM), where a *Large Class* is considered to be a class with NOAM higher than a pre-defined threshold.
- The class number of code lines (LOC), where a *Large Class* is a class with LOC higher than a pre-defined threshold.

Model detection

The detection strategy is then defined as:

LargeClass = (NEC > 20)

Where:

- **Number of Elements in Class (NEC)**: represents the sum of the amount of attributes and methods in a class.

How to pay the debt

Break the class into new classes while NEC is higher than 10.

Name	*Long Parameter List*
Description	
Long Parameter List indicates that a method has too many parameters. Excessive number of parameters in a method is a bad practice because it makes a method more difficult to understand and maintain.	
Source code detection	
Long Parameter List is usually detected in the source code by checking the number of input parameters of a method. When this number is higher than a pre-established threshold, the method is considered an instance of this code smell.	
Model detection	
The detection strategy to find Long Parameter Lists in the models was defined as follows: LongParameterList = (NOP > 6) Where: • **Number Of Parameter (NOP)**: represents the number of parameters of a method	
How to pay the debt	
The main refactoring strategy to pay this debt is to group the parameters that are closely related to each other in a new class so that an instance of this class could be used as input parameter.	

Name	*Shotgun Surgery*

Description

Shotgun Surgery happens when a single change to a class results in many small changes in other classes of the application. This is usually related to high coupled and low cohesive codes.

Source code detection

The strategy to detect *Shotgun Surgery* in the source code is defined by [LANZA, 2010] as follows:

ShotgunSurgery = (CM>10) and ((CM, TopValues(20%)) and (CC>5)

Where:

- **Changing Methods (CM)**: number of methods that could be affected by a change in a class. A method B can be affected by a change in a method A when B uses A somehow. This happens when B extends A, or when B has a method call to A anywhere in its implementation.
- **Changing Classes (CC)**: corresponds to the number of client classes that are potentially affected by changes in one class. A class B can be affected by changes in a class A when B makes references to A's attributes or methods somewhere in its implementation.

Model detection

The detection strategy to find Shotgun Surgery in the models has the following definition:

ShotgunSurgery = (CM+CA>10) and ((CM+CA), TopValues(20%)) and (CC>5)

Where:

- **Changing Methods (CM)**: corresponds to the number of methods possibly affected by a change in one class. It is measured by the number of times that a class is referenced by external methods. In the models, we considered that a method refers to a class when it has one parameter of that class' type or when it overrides a method of that class.
- **Changing Classes (CC)**: corresponds to the number of client classes that may be affected by a change in the measured class. In models, we considered that class B can be affected by some change in class A when B depends on or is associated with A.
- **Changing Attributes (CA)**: corresponds to the number of external classes attributes with the type of a given class. [Isela, 2009] suggested this metric complement the effect of CM metric, since we don't have access to the methods implementation to find all external references it makes.

How to pay the debt

The suggested strategy to pay this debt is to move responsibilities from the methods affected by changes in an instance of *Shotgun Surgery* to methods of the class that has the code smell.

Name	*Data Class*

Description

Data Class corresponds to a class that is used to store data only. As object oriented classes should be an abstraction to both data and behavior, separate behavior from its related data is a bad practice and should be avoided. It usually results in spread responsibilities along the application, which can decrease the code cohesion.

Source code detection

The strategy to detect *Data Class* in the source code is defined by [LANZA, 2010] as follows:

DataClass = (WOC<0.33) and (((NOPA +NOAM>2) and (WMC<31)) or (NOAP+NOAM>7) and (WMC<47))

Where:

- **Weight Of a Class (WOC)**: corresponds to the total number of service methods (methods that are not access only, like getters and setters) divided by the number of interface members of the class. Classes that have too many access methods will have a high value for this metric.
- **Number Of Public Attributes (NOPA)**: corresponds to the number of non-inherited attributes of a class.
- **Number Of Accessor Methods (NOAM)**: corresponds to the number of access methods (getters and setters) that a class contains.
- **Weighted Method Count (WMC)**: corresponds to the sum of the static complexity of all methods of a class.

Model detection

The detection strategy to identify Data Classes in the models was defined as follows:

DataClass = (WOC<0.33) and ((NOAM>2) and WMC<3) or ((NOAM>5) and WMC<4)

Where:

- **Weight Of a Class (WOC)**: have the same meaning as the original source code metric.
- **Number Of Accessor Methods (NOAM)**: have the same meaning as the original source code metric.
- **Weighted Method Count (WMC)**: corresponds to the sum of static complexity of all class methods. In a source code detection strategy, this metric is given by its cyclomatic complexity, for instance. As this information cannot be calculated before code generation, an adaptation was needed for this metric. We used the same adaptation as [ISELA, 2009], both in the metric calculation and the detection strategy thresholds. According to this work, the static complexity of a method can be calculated by using the number of its input parameters. The rationale is that methods that have too many parameters tend to be more complex, then the higher the number of input parameters, the higher is WMC.

How to pay the debt

The main refactoring strategy to pay this debt consists in moving behavior that is related to the attributes of a *Data Class* to methods of the class itself.

Name	*God Class*

Description

God Class happens when a class centralizes a great part of system's intelligence. A *God Class* is usually difficult to reuse and can also hinder system maintainability and understandability.

Source code detection

The strategy defined to detect *God Class* in the source code is defined by [LANZA, 2010] as follows:

GodClasses = (ATFD > 4) and (WMC >Top(20%)) and (TCC < 0.33)

Where:

- **Access To Foreign Data (ATFD)**: represents the number of external classes to which a class has access, either directly or by access methods.
- **Weighted Method Count (WMC)**: corresponds to the sum of the static complexity of all methods of a class.
- **Tight Class Cohesion (TCC)**: corresponds to the relative number of methods that are connected, i.e., that access the same variable of a class. It's a measure of how much cohesive is a class.

Model detection

The detection strategy to find *God Classes* in the models has the following definition:

GodClasses = (ATFD>4) and (NOM>=5) and (CAM<0.33)

Where:

- **Access To Foreign Data (ATFD)**: corresponds to the number of external classes to which one class has access. For EMF models, we consider that class A is accessing a class B if A has an attribute or method parameter of B type.
- **Number Of Methods (NOM)**: corresponds to the number of methods of a class. The original source code detection strategy used the metric Weighted Method Count (WOC), which calculates the methods' static complexity, instead of NOM. As the method's static complexity cannot be calculated with a class model, [Isela, 2009] adapted the use of this WOC to consider that each method has a weight of 1, which leads this metric to be equals to NOM. As they obtained good results with this adaptation, we decided to use the same metric in our catalog.
- **Cohesion Among Methods (CAM)**: calculates the class cohesion based on the number of methods that have the same parameter within a class. The original source code detection strategy uses Tight Class Cohesion (TCC) to calculate class cohesion, but it uses information that is not available in a class model. As with NOM, we decided to use the same adaptation as [Isela, 2009] and replaced this metric by CAM.

How to pay the debt

The main refactoring strategy to pay this debt is to break the *God Class* into smaller and higher cohesive classes.

Name	*Misplaced Class*

Description

Misplaced Class happens when a class relates strongly with classes of other packages than with classes of its own package.

Source code detection

The detection strategy defined by [Lanza, 2010] to find this code smell is the following:

MisplacedClass = (CL<0.33) and (NOED, TopValues(25%)) and (NOED>6) and (DD>3)

Where:

- **Number Of External Dependencies (NOED)**: calculates the number of dependencies that a class have with classes of other packages.
- **Class Locality (CL)**: corresponds to the percentage of dependencies of a class that are with classes of the same package.
- **Dependency Dispersion (DD)**: corresponds to the number of packages on which a class depends.

Model detection

The detection strategy to find *Misplaced Classes* in the models has the following definition:

MisplacedClass = (CL<0.33) and (NOED, TopValues(25%)) and (NOED>6) and (DD>3)

The strategy uses the same metrics defined by the source code detection strategy. Meantime, for the model detection, we considered that a class A depends on a class B when at least one of the following situations is observed:

- Class A is a subclass of class B;
- Class A references class B in one of its attributes;
- Class A has references, as input parameter of some method, class B;

How to pay the debt

The refactoring strategy used pay this debt consists in identifying to which package the *Misplaced Class* is most related. If there is a package with this characteristic, the class must be moved to it. If there is no such package, then the system packages should be reorganized so that

Name	*God Package*

Description

God Package happens when an application package becomes too big and low cohesive.

Source code detection

The strategy defined to detect *God Package* in the source code is defined by [LANZA, 2010] as follows:

> GodPackage = (PS>20) and ((PS, TopValues(25%)) and (NOCC>20) and (NOCP>3) and (PC<0,33)

Where:

- **Package Size (PS)**: corresponds to the number of classes within a package.
- **Number of Client Classes (NOCC)**: represents the number of classes from external packages that references classes of a measured package.
- **Number of Client Packages (NOCP)**: represents the number of external packages that references a measured package.
- **Package Cohesion (PC)**: is defined as the relative number of pairs of dependent classes of a package.

Model detection

The detection strategy to find *God Package* in the models has the following definition:

> GodPackage = (PS>20) and (((PS, TopValues(25%)) and (NOCC'>20) and (NOCP'>3) and (PC'<0,33)

The strategy uses the same metrics defined by the source code detection strategy. Meantime, for the model detection, we considered that a class A depends on a class B when at least one of the following situations is observed:

- Class A is a subclass of class B;
- Class A references class B in one of its attributes;
- Class A has references, as input parameter of some method, class B;

How to pay the debt

The refactoring strategy used to pay this debt consists in reorganizing the system's classes so that the packages become smaller and more cohesive. This can be achieved by moving classes from the *God Package* to other packages or even by breaking it into smaller packages.

Name	*Tradition Breaker*

Description

Tradition Breaker happen when a subclass provides a great amount of services that are not related to its superclass' services.

Source code detection

The strategy defined to detect *Tradition Breaker* in the source code is defined by [LANZA, 2010] as follows:

```
TraditionBreaker = ((NAS >= AVERAGE(NOM)) and (PNAS>=2/3))
            and  (((AMW > AVERAGE) or (WMC >= VERY HIGH)) and (NOM >= HIGH))
            and  ((AMW > AVERAGE) and (NOM > HIGH/2) and (WMC >= VERY HIGH/2)))
```

Where:

- **Newly Added Services (NAS)**: represents the number of public methods added by the subclass.
- **Percentage of Newly Added Services (PNAS)**: represents the percentage growth in the subclass interface compared to the superclass interface.
- **Number of Methods (NOM)**: corresponds to the number of methods of a class.
- **Average Method Weight (AMW)**: corresponds to the average static complexity of all methods of a class.
- **Weighted Method Count (WMC)**: corresponds to the sum of the static complexity of all methods of a class.

Model detection

The detection strategy to find *Tradition Breaker* in the models has the following definition:

```
TraditionBreaker = ((NAS >= AVERAGE(NOM)) and (PNAS>=2/3))
            and (((AMW > AVERAGE) or (WMC >= VERY HIGH)) and (NOM >= HIGH))
            and ((AMW > AVERAGE) and (NOM > HIGH/2) and (WMC >= VERY HIGH/2)))
```

Where:

- **Newly Added Services (NAS)**: same meaning as the original source code metric.
- **Percentage of Newly Added Services (PNAS)**: same meaning as the original source code metric.
- **Number of Methods (NOM)**: same meaning as the original source code metric.
- **Average Method Weight (AMW)**: an adaption of the original source code metric, in which each method is considered to have static complexity equals to 1.
- **Weighted Method Count (WMC)**: an adaption of the original source code metric, in which each method is considered to have static complexity equals to 1.

How to pay the debt

The strategy defined to pay this debt basically consists in assessing the interface of both the superclass and the subclass. Sometimes there are subclass methods that are not used by external clients and then should have their visibility changed from public to protected or private. In other cases there are methods that are implemented in all subclasses and then should be abstracted to the superclass. Sometimes there are so many differences between the interfaces, that there is no real specialization relationship and then the inheritance should be undone.