Commit_hash: SHA checksum of repository revision that the sample was acquired at
Start_line: line in the file in which the sample starts
End_line: line in the file in which the sample ends
Smell: name of the smell (LongMethod, FeatureEnvy, DataClass and Blob)
Id: a numeric identifier of the review
Reviewer_id: a numeric identifier of the reviewer
Sample_id: a numeric identifier of the sample
Severity: severity of the code smell (critical, major, minor, and none)
longmethod_label:checks whether the LongMethod smell severity has at least major, minor or critical
featureenvy_label:checks whether the FeatureEnvy smell severity has at least major, minor or critical
dataclass_label:checks whether the DataClass smell severity has at least major, minor or critical
blob_label:checks whether the Blob smell severity has at least major, minor or critical
agreement_experts:checks whether any smell severity has at least major, minor or critical
agreement_smell_type: checks if at least 2 method (3 tools + MLCQ) found the smell
agreement_smell:checks if at least 2 method (3 tools + MLCQ) found any smell
Review_timestamp: date and time (millisecond precision) when the sample was acquired
Type: whether the reviewed code sample is a class or a function
Code_name: a fully qualified name of the code sample – format: P ackage.ClassN ame[#F unctionN ame arg1|arg2|...] (e.g., org.eclipse.swt.widgets.Menu#setLocation int|int), in case of constructors and static methods a dot is used instead of a hash.
Repository: a git url of the repository
Path: path in the repository that can be used to retrieve the sample
link:a link that can be used to view the sample in a browser.
Is_from_industry_relevant_project:denotes whether source project was classified as industry-relevant in [1]
Weighted_average:
Qtd_rewier_id: number of rewier Identifies
Organic_fileRelativePath: File relative path

,Organic_startLine: line where the smell detected by the tool begins
Organic_endLine: line where the smell detected by the tool ends
,Organic_WeighOfClass:
,Organic_NumberOfAccessorMethods
,Organic_ClassEffectiveLinesOfCode
,Organic_LCOM3:
,Organic_TightClassCohesion,Organic_IsAbstract,Organic_PublicFieldCount,Organic_OverrideRatio,Organic_WeightedMethodCount,
Organic_smellName: Smell Name,
Organic_smellReason:reason that caused the smell,
Organic_ChangingMethods,Organic_NumberOfTryStatements,Organic_MaxNesting,Organic_NumberOfFinallyStatements,Organic_NumberOfThrowStatements,
Organic_NumberOfTryStatementsWithNoCatchAndFinally,Organic_CyclomaticComplexity,Organic_CouplingIntensity,Organic_NumberOfAccessedVariables,Organic_MethodEffectiveLinesOfCode,Organic_CouplingDispersion,Organic_ExceptionalLOC,Organic_NumberOfDummyExceptionHandlers,Organic_ParameterCount,Organic_ChangingClasses,Organic_MaxCallChain,Organic_NumberOfCatchStatements,Organic_ThrownExceptionTypesCount,
PMD_Problem:
PMD_Package:method or class path,
PMD_File:method or class path complete with file name,
PMD_Priority:priority number
PMD_Line: line of the smell detected,
PMD_Description: description of the smell found,
PMD_Rule_set: If the smell found is from Best Practices, code style,design,documentation,Error Prone,Multithreading,Performance, Security or Additional rulesets
PMD_Rule: Smell detected
implementation_Project Name: Project name
Implementation_code_name: method or class path
implementation_Type Name:
implementation_Method Name: name of the method in which the smell was detected
implementation_Method start line no: line where the smell detected by the tool begins
Designite_implementation Smell: Designite smells that occur at the implementation level

implementation_Cause of the Smell: points out the cause of the smell
Designite_Project Name: Project Name
Designite_Package Name: Package Name
Designite_Type Name:
Designite_Design Smell: Designite identifies design smells and presents them in a view that classifies them based on the fundamental principle they violate
Designite_Cause of the Smell: points out the cause of the smell
CK_Class_file:
CK_Class_class:,
CK_Class_type:,
CK_Class_cbo:Counts the number of dependencies a class has. The tools checks for any type used in the entire class (field declaration, method return types, variable declarations, etc). It ignores dependencies to Java itself (e.g. java.lang.String).,
CK_Class_cboModified:Counts the number of dependencies a class has. It is very similar to the CKTool's original CBO. However, this metric considers a dependency from a class as being both the references the type makes to others and the references that it receives from other types.
CK_Class_fanin:Counts the number of input dependencies a class has, i.e, the number of classes that reference a particular class. For instance, given a class X, the fan-in of X would be the number of classes that call X by referencing it as an attribute, accessing some of its attributes, invoking some of its methods, etc.,
CK_Class_fanout:Counts the number of output dependencies a class has, i.e, the number of other classes referenced by a particular class. In other words, given a class X, the fan-out of X is the number of classes called by X via attributes reference, method invocations, object instances, etc.
CK_Class_wmc:It counts the number of branch instructions in a class,
CK_Class_dit:It counts the number of "fathers" a class has. All classes have DIT at least 1 (everyone inherits java.lang.Object). In order to make it happen, classes must exist in the project (i.e. if a class depends upon X which relies in a jar/dependency file, and X depends upon other classes, DIT is counted as 2).
CK_Class_noc: It counts the number of immediate subclasses that a particular class has.
CK_Class_rfc: Counts the number of unique method invocations in a class. As invocations are resolved via static analysis, this implementation fails when a method has overloads with same number of parameters, but different types.,

CK_Class_lcom:Calculates LCOM metric. This is the very first version of metric, which is not reliable. LCOM-HS can be better (hopefully, you will send us a pull request).,
lCK_Class_com*:This metric is a modified version of the current version of LCOM implemented in CK Tool. LCOM* is a normalized metric that computes the lack of cohesion of class within a range of 0 to 1. Then, the closer to 1 the value of LCOM* in a class, the less the cohesion degree of this respective class. The closer to 0 the value of LCOM* in a class, the most the cohesion of this respective class. This implementation follows the third version of LCOM* defined in [1].
,CK_Class_tcc:Measures the cohesion of a class with a value range from 0 to 1. TCC measures the cohesion of a class via direct connections between visible methods, two methods or their invocation trees access the same class variable.
,CK_Class_lcc:Similar to TCC but it further includes the number of indirect connections between visible classes for the cohesion calculation. Thus, the constraint LCC >= TCC holds always.
CK_Class_totalMethodsQty: Counts the number of all methods.
CK_Class_staticMethodsQty: Counts the number of static methods.
CK_Class_publicMethodsQty: Counts the number of public methods
,CK_Class_privateMethodsQty: Counts the number of private methods.
CK_Class_protectedMethodsQty: Counts the number of protected methods.
CK_Class_defaultMethodsQty: Counts the number of default methods.
CK_Class_visibleMethodsQty: Counts the number of visible methods.
CK_Class_abstractMethodsQty: Counts the number of abstract methods.
CK_Class_finalMethodsQty: Counts the number of final methods.
,CK_Class_synchronizedMethodsQty: Counts the number of synchronized methods.
,CK_Class_totalFieldsQty: Counts the number of all fields
,CK_Class_staticFieldsQty: Counts the number of static fields
CK_Class_publicFieldsQty: **Counts the number of public fields**
CK_Class_privateFieldsQty: Counts the number of private fields
,CK_Class_protectedFieldsQty: Counts the number of protected fields
CK_Class_defaultFieldsQty: Counts the number of default fields
CK_Class_finalFieldsQty: Counts the number of final fields
,CK_Class_synchronizedFieldsQty: Counts the number of synchronized fields
CK_Class_nosi: Number of static invocations. Counts the number of invocations to static methods

CK_Class_loc: Lines of code. It counts the lines of the count, ignoring empty lines and comments
CK_Class_returnQty: The number of return instructions
CK_Class_loopQty: The number of loops like for, while, do while and enhanced for
CK_Class_comparisonsQty: The number of comparisons == and !=
CK_Class_tryCatchQty: The number of try/catches
CK_Class_parenthesizedExpsQty: The number of expressions inside parenthesis
CK_Class_stringLiteralsQty: The number of string literals
CK_Class_numbersQty: The number of numbers literals int, long, double, float
CK_Class_assignmentsQty: The number of same or different comparisons
CK_Class_mathOperationsQty: The number of math operations (times, divide, remainder, plus, minus, left shit, right shift)
CK_Class_variablesQty: The number of declared variables
CK_Class_maxNestedBlocksQty: The highest number of blocks nested together
CK_Class_anonymousClassesQty: The quantity of anonymous classes
CK_Class_innerClassesQty: The quantity of inner classes
CK_Class_lambdasQty: The quantity of lambda expressions
CK_Class_uniqueWordsQty: The algorithm basically counts the number of words in a class, after removing Java keywords
CK_Class_modifiers:public/abstract/private/protected/native modifiers of classes/methods. Can be decoded using org.eclipse.jdt.core.dom.Modifier,
CK_Class_logStatementsQty: Number of log statements in the source code,
,CK_Method_file: file name,
CK_Method_class: Class of method,
CK_Method_method: Method name,
CK_Method_constructor: Constructor name,
CK_Method_line: Method line
,CK_Method_cbo: Counts the number of dependencies a class has. The tools checks for any type used in the entire class (field declaration, method return types, variable declarations, etc). It ignores dependencies to Java itself (e.g. java.lang.String).,
CK_Method_cboModified:Counts the number of dependencies a class has. It is very similar to the CKTool's original CBO. However, this metric considers a dependency from a class as being both the references the type makes to others and the references that it receives from other types.,

CK_Method_fanin:Counts the number of input dependencies a class has, i.e, the number of classes that reference a particular class. For instance, given a class X, the fan-in of X would be the number of classes that call X by referencing it as an attribute, accessing some of its attributes, invoking some of its methods, etc.,
CK_Method_fanout: Counts the number of output dependencies a class has, i.e, the number of other classes referenced by a particular class. In other words, given a class X, the fan-out of X is the number of classes called by X via attributes reference, method invocations, object instances, etc.,
CK_Method_wmc:It counts the number of branch instructions in a class.,t counts the number of branch instructions in a class.
,CK_Method_rfc:Counts the number of unique method invocations in a class. As invocations are resolved via static analysis, this implementation fails when a method has overloads with same number of parameters, but different types.,
CK_Method_loc: It counts the lines of count, ignoring empty lines and comments (i.e., it's Source Lines of Code, or SLOC). The number of lines here might be a bit different from the original file, as we use JDT's internal representation of the source code to calculate it.,
CK_Method_returnsQty:The number of return instructions.,
CK_Method_variablesQty: Number of declared variables.,
CK_Method_parametersQty: ,
CK_Method_methodsInvokedQty:All directly invoked methods, variations are local invocations and indirect local invocations.,
CK_Method_methodsInvokedLocalQty:All directly invoked methods, variations are local invocations and indirect local invocations.,
CK_Method_methodsInvokedIndirectLocalQty,:All directly invoked methods, variations are local invocations and indirect local invocations.
CK_Method_loopQty: The number of loops ,
CK_Method_comparisonsQty: The number of comparisons,
CK_Method_tryCatchQty: The number of try/catches,
CK_Method_parenthesizedExpsQty The number of expressions inside parenthesis.,
CK_Method_stringLiteralsQty: The number of string literals,
CK_Method_numbersQty: *Quantity of Number*,
CK_Method_assignmentsQty: The number of same or different comparisons ,
CK_Method_mathOperationsQty: The number of math operations ,
CK_Method_maxNestedBlocksQty:The highest number of blocks nested together.,

CK_Method_anonymousClassesQty: The quantity of anonymous classes,
CK_Method_innerClassesQty: The quantity of inner classes,
CK_Method_lambdasQty: The quantity of lambda expressions ,
CK_Method_uniqueWordsQty: Number of unique words in the source code.,
CK_Method_modifiers:public/abstract/private/protected/native modifiers of classes/methods. Can be decoded using org.eclipse.jdt.core.dom.Modifier,
CK_Method_logStatementsQty: Number of log statements in the source code,
CK_Method_hasJavaDoc: Boolean indicating whether a method has javadoc. (Only at method-level for now),

'system'
'Stars'
'Watching'
'Commits'
'LOC'
'Number of Contributors'
'cluster_kmeans_All'
'cluster_Gaussian_All'
'cluster_kmeans_LOC'
'cluster_Gaussian_LOC'

'smell_Designite': Merge between Designite_implementation Smell column with Designite_Design Smell column
'smell_Organic_aglomeration': the tool organic automatically classified more than one smell (independently of the type)
'smell_PMD_aglomeration': the tool PMD automatically classified more than one smell (independently of the type)
'smell_Designite_aglomeration': the tool Designite automatically classified more than one smell (independently of the type)
'smell_Organic_num_aglomeration'
'smell_PMD_num_aglomeration'
'smell_Designite_num_aglomeration'
'smell_Organic_longmethod',
'smell_Organic_featureenvy'
'smell_Organic_dataclass'
'smell_Organic_blob'

'smell_PMD_longmethod'
'smell_PMD_featureenvy'
'smell_PMD_dataclass'
'smell_PMD_blob'
'smell_Designite_longmethod'
'smell_Designite_featureenvy'
'smell_Designite_dataclass'
'smell_Designite_blob'
'background'