CodeNarc Report

Report title:	Sample Report
Date:	28 nov. 2013 21:45:55
Generated with:	CodeNarc v0.19

Summary by Package

Package	Total Files	Files with Violations	Priority 1	Priority 2	Priority 3
All Packages	25	14	-	**23**	**1**
grails-app/controllers/pastamanga	8	1	-	**1**	-
grails-app/domain/pastamanga	6	4	-	**8**	-
test/unit/pastamanga	11	9	-	**14**	**1**

Package: grails-app.controllers.pastamanga

➥ LoginController.groovy

Rule Name	Priority	Line #	Source Line / Message
UnusedVariable	**2**	92	[SRC]def username = session[UsernamePasswordAuthenticationFil..SERNAME_KEY] [MSG]*The variable [username] in class pastamanga.LoginController is not used*

Package: grails-app.domain.pastamanga

➥ Anime.groovy

Rule Name	Priority	Line #	Source Line / Message
EqualsAndHashCode	2	4	[SRC]class Anime { [MSG]*The class pastamanga.Anime defines equals(Object) but not hashCode()*
DeadCode	2	24	[SRC]+ date_added + '', description='' + description + '']; [MSG]*This code cannot be reached*

➥ Episode.groovy

Rule Name	Priority	Line #	Source Line / Message
GrailsDomainHasEquals	2	3	[SRC]class Episode { [MSG]*The domain class pastamanga.Episode should define an equals(Object) method*
GrailsDomainHasToString	2	3	[SRC]class Episode { [MSG]*The domain class pastamanga.Episode should define a toString() method*

➥ User.groovy

Rule Name	Priority	Line #	Source Line / Message
GrailsDomainHasEquals	2	3	[SRC]class User { [MSG]*The domain class pastamanga.User should define an equals(Object) method*
GrailsDomainHasToString	2	3	[SRC]class User { [MSG]*The domain class pastamanga.User should define a toString() method*
GrailsDomainWithServiceReference	2	5	[SRC]transient springSecurityService [MSG]*Violation in class pastamanga.User. Domain class User should not*

			reference services (offending field: springSecurityService)

➥ UserRole.groovy

Rule Name	Priority	Line #	Source Line / Message
GrailsDomainHasToString	2	5	[SRC]class UserRole implements Serializable { [MSG]*The domain class pastamanga.UserRole should define a toString() method*

Package: test.unit.pastamanga

➥ AnimeControllerSpec.groovy

Rule Name	Priority	Line #	Source Line / Message
UnusedImport	3	7	[SRC]import javax.validation.constraints.AssertTrue; [MSG]*The [javax.validation.constraints.AssertTrue] import is never referenced*

➥ AnimeSpec.groovy

Rule Name	Priority	Line #	Source Line / Message
UnusedVariable	2	14	[SRC]def Anime01 = new Anime(name: "test01", category: "scar"..on") == true [MSG]*The variable [Anime01] in class pastamanga.AnimeSpec is not used*

➥ ApiControllerSpec.groovy

Rule Name	Priority	Line #	Source Line / Message
EmptyMethod	2	12	[SRC]def setup() { [MSG]*Violation in class ApiControllerSpec. The method setup is both empty and not marked with @Override*
EmptyMethod	2	16	[SRC]def cleanup() {

			[MSG]*Violation in class ApiControllerSpec. The method cleanup is both empty and not marked with @Override*
EmptyMethod	**2**	19	[SRC]void "test something"() { [MSG]*Violation in class ApiControllerSpec. The method test something is both empty and not marked with @Override*

➥ EpisodeControllerSpec.groovy

Rule Name	Priority	Line #	Source Line / Message
EmptyMethod	**2**	12	[SRC]def setup() { [MSG]*Violation in class EpisodeControllerSpec. The method setup is both empty and not marked with @Override*
EmptyMethod	**2**	15	[SRC]def cleanup() { [MSG]*Violation in class EpisodeControllerSpec. The method cleanup is both empty and not marked with @Override*
EmptyMethod	**2**	18	[SRC]void "test something"() { [MSG]*Violation in class EpisodeControllerSpec. The method test something is both empty and not marked with @Override*

➥ EpisodeSpec.groovy

Rule Name	Priority	Line #	Source Line / Message
UnusedVariable	**2**	14	[SRC]def Episode01 = new Episode(name: "EpisodeTest01",nameAn..") == true [MSG]*The variable [Episode01] in class pastamanga.EpisodeSpec is not used*

➥ SecureControllerSpec.groovy

Rule Name	Priority	Line #	Source Line / Message
EmptyMethod	**2**	12	[SRC]def setup() { [MSG]*Violation in class SecureControllerSpec. The method setup is both empty and not marked with @Override*
EmptyMethod	**2**	15	[SRC]def cleanup() { [MSG]*Violation in class SecureControllerSpec. The method cleanup is both empty and not marked with*

			@Override
EmptyMethod	**2**	18	[SRC]void "test something"() { [MSG]*Violation in class SecureControllerSpec. The method test something is both empty and not marked with @Override*

➥ UserAnimeSpec.groovy

Rule Name	Priority	Line #	Source Line / Message
UnusedVariable	**2**	16	[SRC]def UserAnime01 = new UserAnime(user: User01, anime: Anime01) == true; [MSG]*The variable [UserAnime01] in class pastamanga.UserAnimeSpec is not used*

➥ UserRoleSpec.groovy

Rule Name	Priority	Line #	Source Line / Message
UnusedVariable	**2**	16	[SRC]def UserRole01 = new UserRole(user: User01,role: Role01) == true [MSG]*The variable [UserRole01] in class pastamanga.UserRoleSpec is not used*

➥ UserSpec.groovy

Rule Name	Priority	Line #	Source Line / Message
UnusedVariable	**2**	14	[SRC]def User01 = new User(username: "UserTest01", password: ..lse) == true [MSG]*The variable [User01] in class pastamanga.UserSpec is not used*

Rule Descriptions

#	Rule Name	Description
1	**AssertWithinFinallyBlock**	Checks for ***assert*** statements within a ***finally*** block. An ***assert*** can throw an exception, hiding the original exception, if there is one.
2	**AssignmentInConditional**	An assignment operator (=) was used in a conditional test. This is usually a typo, and the comparison operator (==) was intended.

3	**BigDecimalInstantiation**	Checks for calls to the ***BigDecimal*** constructors that take a ***double*** parameter, which may result in an unexpected ***BigDecimal*** value.
4	**BitwiseOperatorInConditional**	Checks for bitwise operations in conditionals, if you need to do a bitwise operation then it is best practive to extract a temp variable.
5	**BooleanGetBoolean**	This rule catches usages of java.lang.Boolean.getBoolean(String) which reads a boolean from the System properties. It is often mistakenly used to attempt to read user input or parse a String into a boolean. It is a poor piece of API to use; replace it with System.properties['prop'].
6	**BrokenNullCheck**	Looks for faulty checks for null that can cause a ***NullPointerException***.
7	**BrokenOddnessCheck**	The code uses x % 2 == 1 to check to see if a value is odd, but this won't work for negative numbers (e.g., (-5) % 2 == -1). If this code is intending to check for oddness, consider using x & 1 == 1, or x % 2 != 0.
8	**CatchArrayIndexOutOfBounds Exception**	Check the size of the array before accessing an array element rather than catching ***ArrayIndexOutOfBoundsException***.
9	**CatchError**	Catching ***Error*** is dangerous; it can catch exceptions such as ***ThreadDeath*** and ***OutOfMemoryError***.
10	**CatchException**	Catching ***Exception*** is often too broad or general. It should usually be restricted to framework or infrastructure code, rather than application code.
11	**CatchIllegalMonitorStateExcept ion**	Dubious catching of IllegalMonitorStateException. IllegalMonitorStateException is generally only thrown in case of a design flaw in your code (calling wait or notify on an object you do not hold a lock on).
12	**CatchIndexOutOfBoundsExcept ion**	Check that an index is valid before accessing an indexed element rather than catching ***IndexOutOfBoundsException***.
13	**CatchNullPointerException**	Catching ***NullPointerException*** is never appropriate. It should be avoided in the first place with proper null checking, and it can mask underlying errors.
14	**CatchRuntimeException**	Catching ***RuntimeException*** is often too broad or general. It should usually be restricted to framework or infrastructure code, rather than application code.
15	**CatchThrowable**	Catching ***Throwable*** is dangerous; it can catch exceptions such as ***ThreadDeath*** and ***OutOfMemoryError***.
16	**ClassForName**	Using Class.forName(...) is a common way to add dynamic behavior to a system. However, using this method can cause resource leaks because the classes can be pinned in memory for long periods of time.
17	**ComparisonOfTwoConstants**	Checks for expressions where a comparison operator or ***equals()*** or ***compareTo()*** is used to compare two constants to each other or two literals that contain only

		constant values., e.g.: ***23 == 67, Boolean.FALSE != false, 0.17 <= 0.99, "abc" > "ddd", [a:1] <=> [a:2], [1,2].equals([3,4]) or [a:false, b:true].compareTo(['a':34.5, b:Boolean.TRUE]***.
18	**ComparisonWithSelf**	Checks for expressions where a comparison operator or ***equals()*** or ***compareTo()*** is used to compare a variable to itself, e.g.: ***x == x, x != x, x <=> x, x < x, x =>= x, x.equals(x) or x.compareTo(x)***, where ***x*** is a variable.
19	**ConfusingClassNamedException**	This class is not derived from another exception, but ends with 'Exception'. This will be confusing to users of this class.
20	**ConstantAssertExpression**	Checks for ***assert*** statements where the assert boolean condition expression is a constant or literal value.
21	**ConstantIfExpression**	Checks for ***if*** statements with a constant value for the if expression, such as ***true***, ***false***, ***null***, or a literal constant value.
22	**ConstantTernaryExpression**	Checks for ternary expressions with a constant value for the boolean expression, such as ***true***, ***false***, ***null***, or a literal constant value.
23	**DeadCode**	Dead code appears after a return statement or an exception is thrown. If code appears after one of these statements then it will never be executed and can be safely deleted.
24	**DoubleNegative**	There is no point in using a double negative, it is always positive. For instance ***!!x*** can always be simplified to x. And ***!(!x)*** can as well.
25	**DuplicateCaseStatement**	Check for duplicate ***case*** statements in a ***switch*** block, such as two equal integers or strings.
26	**DuplicateImport**	Duplicate import statements are unnecessary.
27	**DuplicateMapKey**	A map literal is created with duplicated key. The map entry will be overwritten.
28	**DuplicateSetValue**	A Set literal is created with duplicate constant value. A set cannot contain two elements with the same value.
29	**EmptyCatchBlock**	In most cases, exceptions should not be caught and ignored (swallowed).
30	**EmptyClass**	Reports classes without methods, fields or properties. Why would you need a class like this?
31	**EmptyElseBlock**	Empty ***else*** blocks are confusing and serve no purpose.
32	**EmptyFinallyBlock**	Empty ***finally*** blocks are confusing and serve no purpose.
33	**EmptyForStatement**	Empty ***for*** statements are confusing and serve no purpose.
34	**EmptyIfStatement**	Empty ***if*** statements are confusing and serve no purpose.

35	**EmptyInstanceInitializer**	An empty class instance initializer was found. It is safe to remove it.
36	**EmptyMethod**	A method was found without an implementation. If the method is overriding or implementing a parent method, then mark it with the @Override annotation.
37	**EmptyStaticInitializer**	An empty static initializer was found. It is safe to remove it.
38	**EmptySwitchStatement**	Empty ***switch*** statements are confusing and serve no purpose.
39	**EmptySynchronizedStatement**	Empty ***synchronized*** statements are confusing and serve no purpose.
40	**EmptyTryBlock**	Empty ***try*** blocks are confusing and serve no purpose.
41	**EmptyWhileStatement**	Empty ***while*** statements are confusing and serve no purpose.
42	**EqualsAndHashCode**	If either the ***boolean equals(Object)*** or the ***int hashCode()*** methods are overridden within a class, then both must be overridden.
43	**EqualsOverloaded**	The class has an equals method, but the parameter of the method is not of type Object. It is not overriding equals but instead overloading it.
44	**ExceptionExtendsError**	Errors are system exceptions. Do not extend them.
45	**ExceptionNotThrown**	Checks for an exception constructor call without a ***throw*** as the last statement within a catch block.
46	**ExplicitGarbageCollection**	Calls to System.gc(), Runtime.getRuntime().gc(), and System.runFinalization() are not advised. Code should have the same behavior whether the garbage collection is disabled using the option -Xdisableexplicitgc or not. Moreover, "modern" jvms do a very good job handling garbage collections. If memory usage issues unrelated to memory leaks develop within an application, it should be dealt with JVM options rather than within the code itself.
47	**ForLoopShouldBeWhileLoop**	A for loop without an init and update statement can be simplified to a while loop.
48	**GrailsDomainHasEquals**	Checks that Grails domain classes redefine equals().
49	**GrailsDomainHasToString**	Checks that Grails domain classes redefine toString()
50	**GrailsDomainReservedSqlKeywordName**	Forbids usage of SQL reserved keywords as class or field names in Grails domain classes. Naming a domain class (or its field) with such a keyword causes SQL schema creation errors and/or redundant table/column name mappings.
51	**GrailsDomainWithServiceReference**	Checks that Grails domain classes do not have service classes injected.

52	**GrailsDuplicateConstraint**	Check for duplicate entry in domain class constraints
53	**GrailsDuplicateMapping**	Check for duplicate name in a domain class mapping
54	**GrailsServletContextReference**	Checks for references to the ***servletContext*** object from within Grails controller and taglib classes.
55	**GrailsStatelessService**	Checks for fields on Grails service classes. Grails service classes are singletons, by default, and so they should be reentrant and typically stateless. The ***ignoreFieldNames*** property (dataSource,scope,sessionFactory,transactional,*Service) specifies one or more field names that should be ignored. The ***ignoreFieldTypes*** property (null) specifies one or more field type names that should be ignored. Both can optionally contain wildcard characters ('*' or '?').
56	**HardCodedWindowsFileSeparator**	This rule finds usages of a Windows file separator within the constructor call of a File object. It is better to use the Unix file separator or use the File.separator constant.
57	**HardCodedWindowsRootDirectory**	This rule find cases where a ***File*** object is constructed with a windows-based path. This is not portable, and using the ***File.listRoots()***method is a better alternative.
58	**ImportFromSamePackage**	An import of a class that is within the same package is unnecessary.
59	**ImportFromSunPackages**	Avoid importing anything from the 'sun.*' packages. These packages are not portable and are likely to change.
60	**IntegerGetInteger**	This rule catches usages of java.lang.Integer.getInteger(String, ...) which reads an Integer from the System properties. It is often mistakenly used to attempt to read user input or parse a String into an Integer. It is a poor piece of API to use; replace it with System.properties['prop'].
61	**MisorderedStaticImports**	Static imports should never be declared after nonstatic imports.
62	**MissingNewInThrowStatement**	A common Groovy mistake when throwing exceptions is to forget the new keyword. For instance, "throw RuntimeException()" instead of "throw new RuntimeException()". If the error path is not unit tested then the production system will throw a Method Missing exception and hide the root cause. This rule finds constructs like "throw RuntimeException()" that look like a new keyword was meant to be used but forgotten.
63	**RandomDoubleCoercedToZero**	The Math.random() method returns a double result greater than or equal to 0.0 and less than 1.0. If you coerce this result into an Integer or int, then it is coerced to zero. Casting the result to int, or assigning it to an int field is probably a bug.

64	**RemoveAllOnSelf**	Don't use ***removeAll*** to clear a collection. If you want to remove all elements from a collection c, use c.clear, not c.removeAll(c). Calling***c.removeAll(c)*** to clear a collection is less clear, susceptible to errors from typos, less efficient and for some collections, might throw a***ConcurrentModificationException***.
65	**ReturnFromFinallyBlock**	Returning from a ***finally*** block is confusing and can hide the original exception.
66	**ReturnNullFromCatchBlock**	Returning ***null*** from a catch block often masks errors and requires the client to handle error codes. In some coding styles this is discouraged.
67	**SwallowThreadDeath**	Checks for code that catches ThreadDeath without re-throwing it.
68	**ThrowError**	Checks for throwing an instance of ***java.lang.Error***.
69	**ThrowException**	Checks for throwing an instance of ***java.lang.Exception***.
70	**ThrowExceptionFromFinallyBlock**	Throwing an exception from a ***finally*** block is confusing and can hide the original exception.
71	**ThrowNullPointerException**	Checks for throwing an instance of ***java.lang.NullPointerException***.
72	**ThrowRuntimeException**	Checks for throwing an instance of ***java.lang.RuntimeException***.
73	**ThrowThrowable**	Checks for throwing an instance of ***java.lang.Throwable***.
74	**UnnecessaryGroovyImport**	A Groovy file does not need to include an import for classes from ***java.lang***, ***java.util***, ***java.io***, ***java.net***, ***groovy.lang*** and ***groovy.util***, as well as the classes ***java.math.BigDecimal*** and ***java.math.BigInteger***.
75	**UnusedArray**	Checks for array allocations that are not assigned or used, unless it is the last statement within a block.
76	**UnusedImport**	Imports for a class that is never referenced within the source file is unnecessary.
77	**UnusedMethodParameter**	This rule finds instances of method parameters not being used. It does not analyze private methods (that is done by the UnusedPrivateMethodParameter rule) or methods marked @Override.
78	**UnusedObject**	Checks for object allocations that are not assigned or used, unless it is the last statement within a block
79	**UnusedPrivateField**	Checks for private fields that are not referenced within the same class.
80	**UnusedPrivateMethod**	Checks for private methods that are not referenced within the same class.
81	**UnusedPrivateMethodParameter**	Checks for parameters to private methods that are not referenced within the method body.

82	**UnusedVariable**	Checks for variables that are never referenced. The ***ignoreVariableNames*** property (null) specifies one or more variable names that should be ignored, optionally containing wildcard characters ('*' or '?').