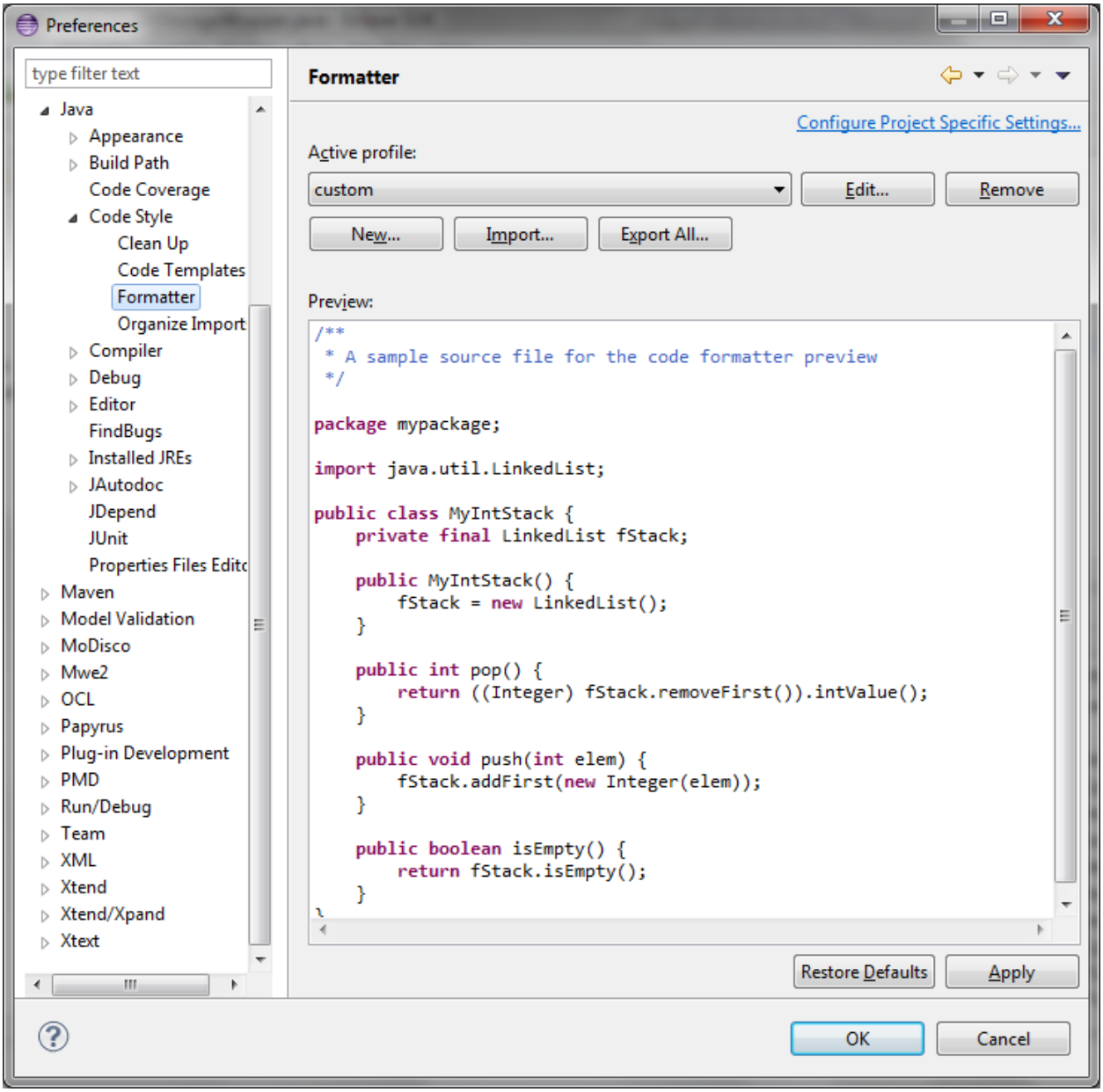
# Eclipse Preferences

Eclipse core preferences – This file is present in Git under location : eclipse_script/ MyPrefs.epf. This is a **workspace specific** setting I think and the file needs to be imported inside eclipse every time we create a new workspace.

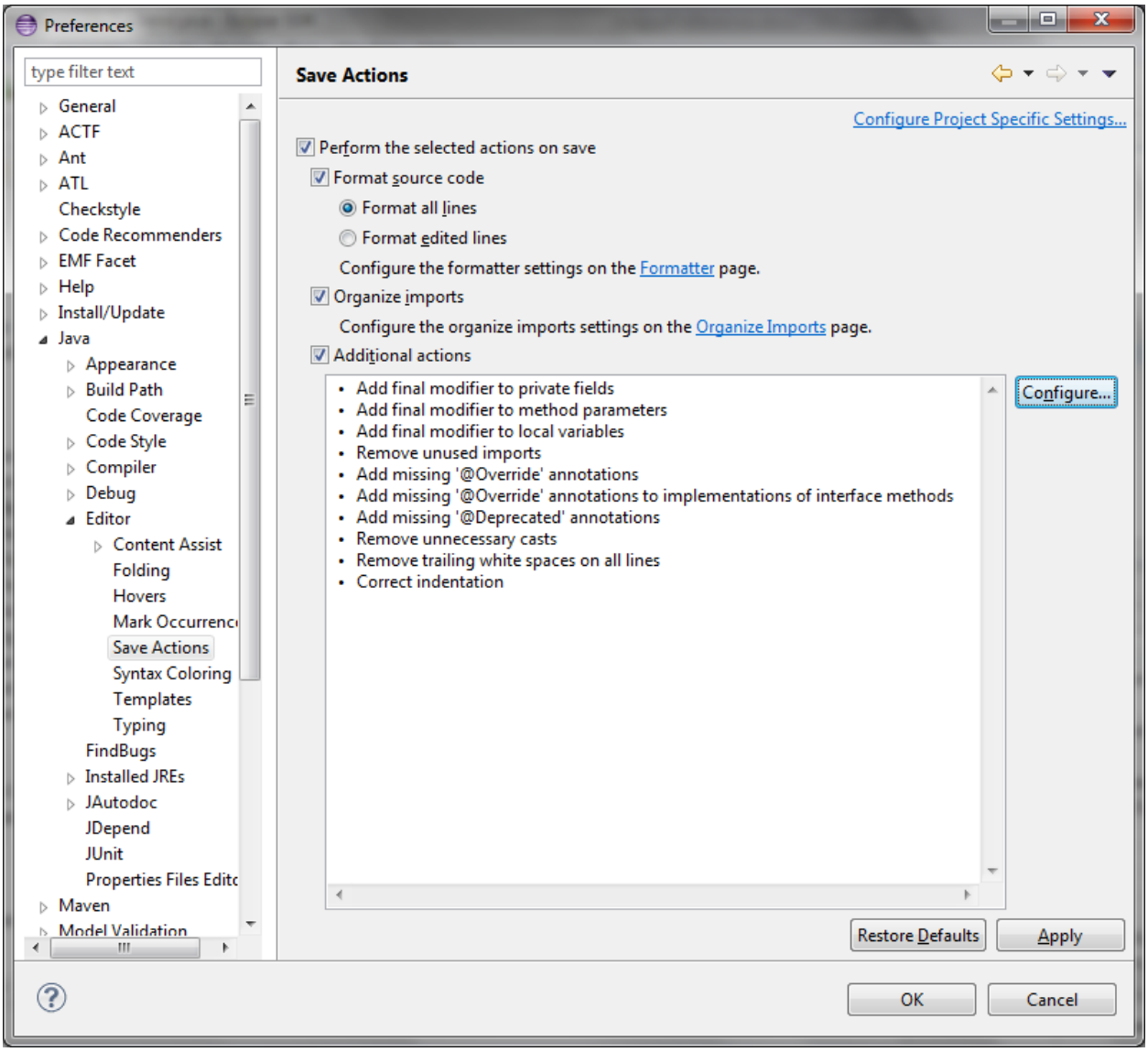
**Code Formatter Preferences** - Created a custom preferences which is present in Git under location : eclipse_script/custom_code_formatter.xml. This is a **workspace specific** setting I think and the file needs to be imported inside eclipse every time we create a new workspace. Following screenshot shows how to import this file.



Comments:

- This is an extension of in-built “Java Conventions” formatter preferences.
- There is only one difference in “Java Conventions” formatter preferences and “Custom Preferences”

Changed “Save Actions” Eclipse Preferences – These preferences will be applied when we import the “MyPrefs.epf” file as mentioned in step 1.

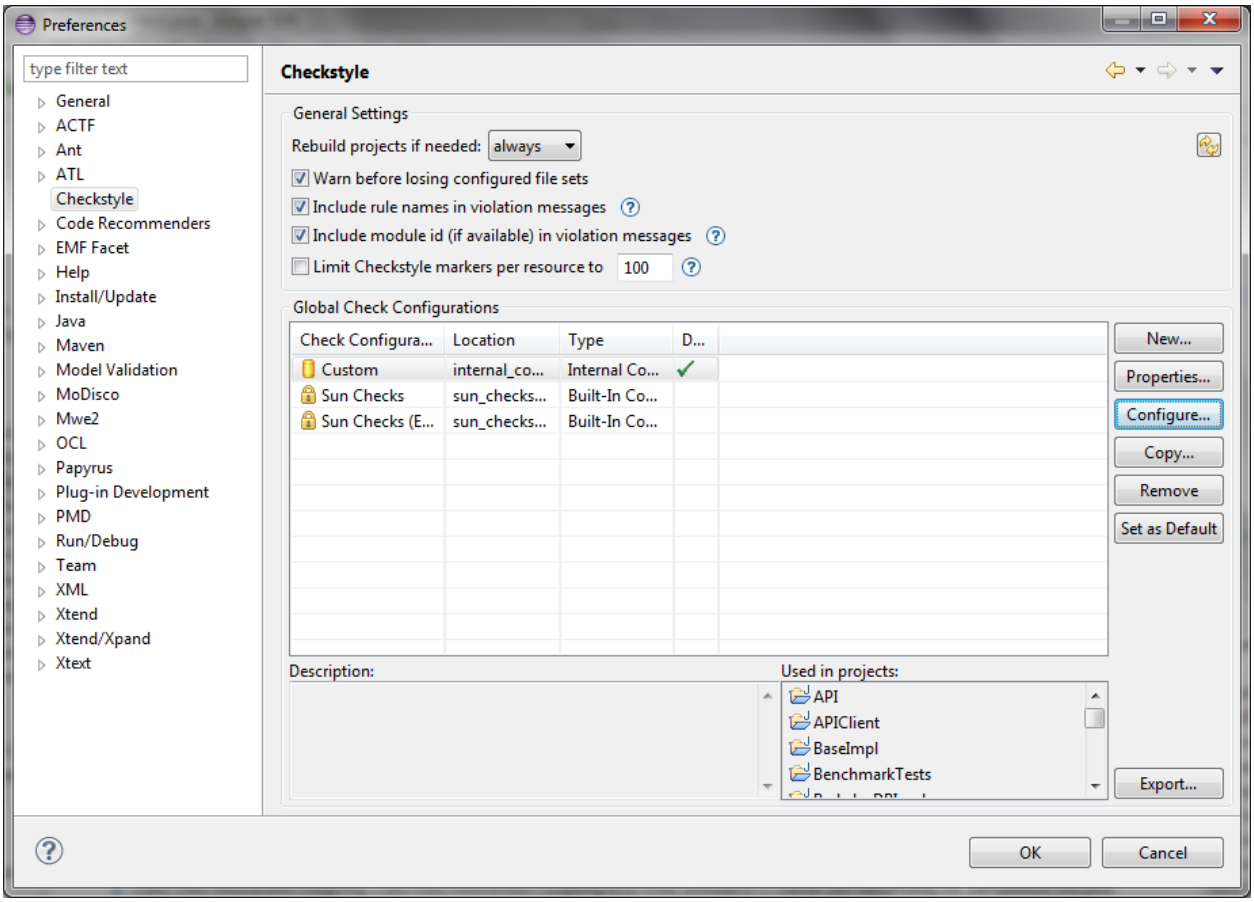



Comments:

- Auto formats the code when file is saved.
- Makes all the method parameters ‘final’ wherever possible.
- Makes local variables ‘final’ wherever possible.

- Please check the above screenshot for more details.

**Checkstyle Preferences** – File eclipse_script/checkstyle.xml has the custom preference related to checkstyle plugin and these preferences should be applied separately at workspace level.




Comments:

- Naming convention relate rules :
  - Changed the rule so that all the 'static' variables are 'All Caps"
  - All the other naming conventions are java standard naming conventions.
- Javadoc Comments-
  - Checks if Javadoc comments are presents for public and protected members.
  - Other restrictions within javadocs like checking if @param tag exists are disabled.
- Size Violation – Severity level changed to **'Info'**
  - Max file length – 2000 lines
  - Max method length 150 lines.
  - Max number of Parameters 7.
- Some important Rules that I have **enabled** :
  - Checks if Checks that the order of modifiers conforms to the suggestions in the Java Language specification.

- Checks for redundant modifiers. (Info Level.)
- Checks that classes that define a covariant equals() method also override method equals(java.lang.Object). Inspired by findbugs.
- Check that the `default` is after all the cases in a `switch` statement.
- Detects empty statements (standalone ';' ) (info level).
- Checks that classes that override `equals()` also override `hashCode()`.
- Checks that any combination of String literals with optional assignment is on the left side of an equals() comparison. The check also processes `String.equalsIgnoreCase()` invocations (which can be suppressed).
- Checks if any class or object member explicitly initialized to default for its type value (`null` for object references, zero for numeric types and `char` and `false` for `boolean`.
- Checks for fall through in switch statements Finds locations where a `case` contains Java code - but lacks a `break, return, throw` or `continue` statement.
- Checks that local variables that never have their values changed are declared final. The check can be configured to also check that unchanged parameters are declared final.
- Checks can be used to ensure that types are not declared to be thrown. Declaring to throw java.lang.Error or java.lang.RuntimeException is almost never acceptable. Is this needed or should be allow to throw RuntimeException ?
- Checks that particular classes are never used as types in variable declarations (Like ArrayList, HashMap etc), return values or parameters. Includes a pattern check that by default disallows abstract classes.
- Checks that switch statement has "default" clause.
- Check for ensuring that for loop control variables are not modified inside the for block. (Changed to **Info** level)
- Checks that each variable declaration is in its own statement and on its own line. Multiple declarations in same line not allowed e.g. (int i, j ;)
- Checks there is only one statement per line. The following line will be flagged as an error: `x = 1; y = 2; // Two statments on a single line.`
- Checks for redundant exceptions declared in throws clause such as duplicates, unchecked exceptions or subclasses of another declared exception.
- Checks that string literals are not used with == or !=.
- Checks that a class which has only private constructors is declared as final
- Make sure that utility classes (classes that contain only static methods or fields in their API) do not have a public constructor.
- Boolean expression complexity - Restrict the number of number of `&&, ||, &, |` and `^` in an expression.
- Measures the number of instantiations of other classes within the given class. Current threshold is 7 and severity level is **info**
- The number of other classes a given class relies on. Current threshold is 20 and severity level is **info.**
- Checks cyclomatic complexity against a specified limit. The complexity is measured by the number of if, while, do, for, ?:, catch, switch, case statements, and operators && and || (plus one) in the body of a constructor, method, static initializer, or instance initializer. It is a measure of the minimum number of possible paths through the source and therefore the number of required tests. Generally 1-4 is considered good, 5-7 ok, 8-10 consider re-factoring, and 11+ re-factor now! Current threshold is 10 and severity level is **info.**
- Determines complexity of methods, classes and files by counting the Non Commenting Source Statements (NCSS). Current thresholds are method-max 50, class-max 1500, file-max 2000. Severity level is **info**
- The NPATH metric computes the number of possible execution paths through a function. It takes into account the nesting of conditional statements and multi-part boolean expressions (e.g., A && B, C || D, etc.). Current threshold is 200 and severity level is **info.**

- Check that method/constructor parameters are final. Interface methods are not checked - the final keyword does not make sense for interface method parameters as there is no code that could modify the parameter. Severity level is **info.**
- Checks for uncommented main() methods (debugging leftovers). Severity level is **info.**
- Checks that long constants are defined with an upper ell. That is ' L' and not 'l'. This is in accordance to the Java Language Specification, Section 3.10.1. Rationale: The lower case letter l looks a lot like the digit 1.

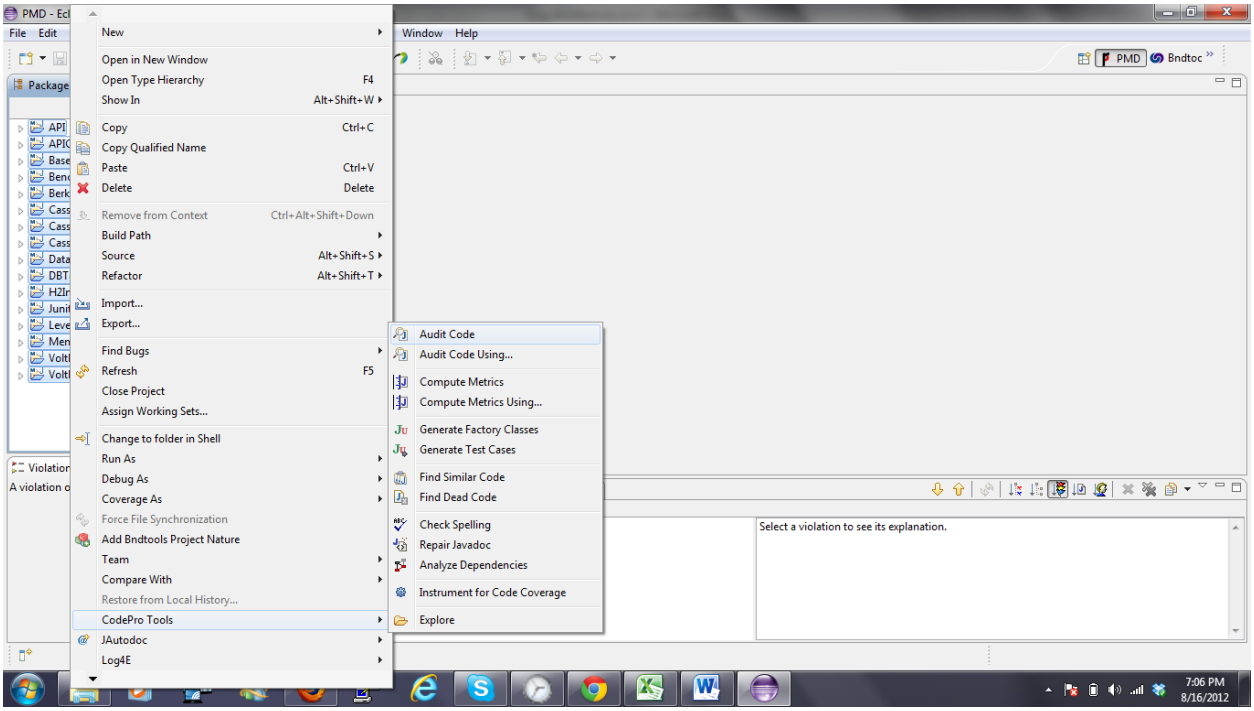
- Rules that I have **not enabled** :
  - Duplicate code - Performs a line-by-line comparison of all code lines and reports duplicate code if a sequence of lines differs only in indentation. (threshold value is 8 lines)
  - Checks that there are no static import statements.
  - Checks the ordering/grouping of imports. Ensures that groups of imports come in a specific order (e.g., java. comes first, javax. comes first, then everything else) and imports within each group are in lexicographic order. Static imports must be at the end of a group and in lexicographic order amongst themselves.
  - Checks the number of methods declared in each type. E.g. max public methods in a class etc.
  - Detects inline conditionals. An example inline conditional is this:
    ```
    String b = (a==null || a.length<1) ? null : a.substring(1);
    ```
  - Declaration order check - According to Code Conventions for the Java Programming Language , the parts of a class or interface declaration should appear in the following order:

    1. Class (static) variables. First the public class variables, then the protected, then package level (no access modifier), and then the private.
    2. Instance variables. First the public class variables, then the protected, then package level (no access modifier), and then the private.
    3. Constructors
    4. Methods

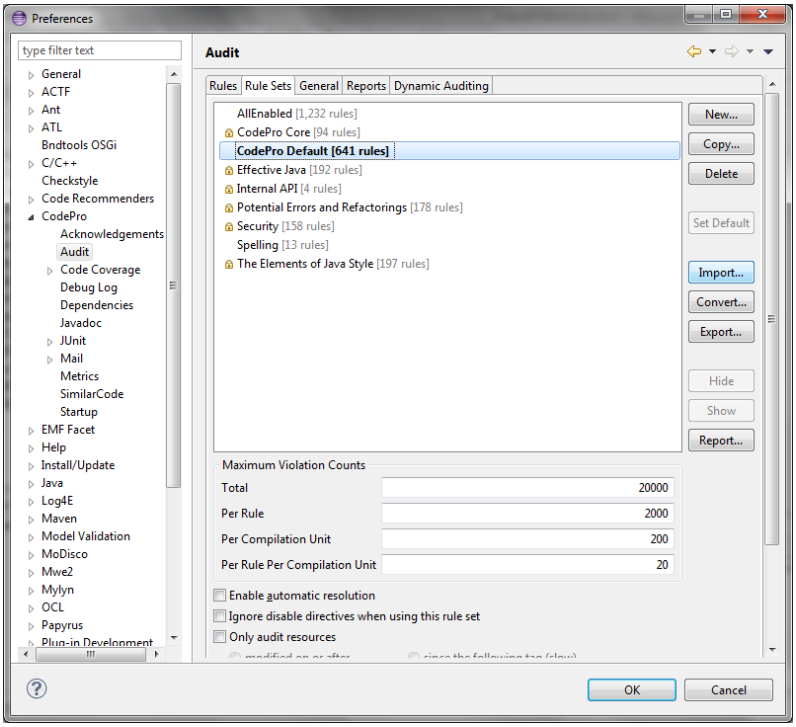
  - Checks that a local variable or a parameter does not shadow a field that is defined in the same class.
  - Checks that there are no "magic numbers", where a magic number is a numeric literal that is not defined as a constant. By default, -1, 0, 1, and 2 are not considered to be magic numbers.
  - Checks for multiple occurrences of the same string literal within a single file.
  - Design For Extension - Checks that classes are designed for extension. More specifically, it enforces a programming style where superclasses provide empty "hooks" that can be implemented by subclasses.
  - Interface is a Type – "it's inappropriate to define an interface that does not contain any methods but only constants", Use Interfaces only to define types.
  - Checks visibility of class members. Only static final members may be public; other class members must be private.
  - Restricts throws statements to a specified count (default = 1).
  - Checks the style of array type definitions. Some like Java-style: public static void main(String[] args) and some like C-style: public static void main(String args[]).
  - Checks whether files end with a new line.
    Rationale: Any source files and text files in general should end with a newline character, especially when using SCM systems such as CVS. CVS will even print a warning when it encounters a file that doesn't end with a newline.
  - The check to ensure that requires that comments be the only thing on a line. For the case of // comments that means that the only thing that should precede it is whitespace.

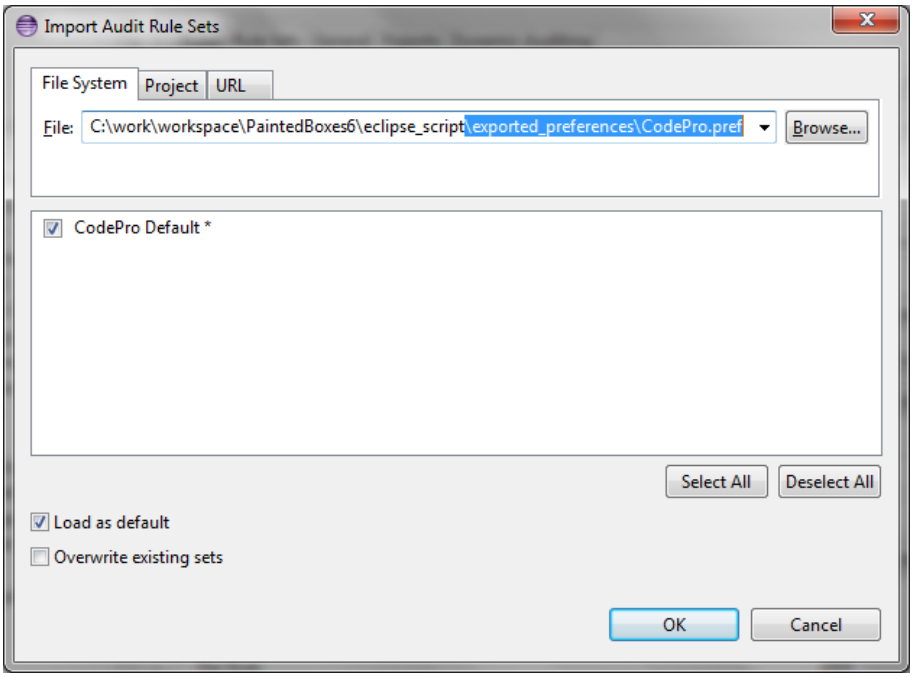
# Code Pro Rules:

## Running the CodePro static analyzer:




## Importing Rules (If template workspace was not used.)






## List of rules enabled/disabled:

(Please check the Appendix for more details about the rules)

| Rule | Enabled | Reason |
|---|---|---|
| 'ANYONE' Method Permission | No | not relevant |
| Abstract Specialization | Yes | Enabled |
| Accessor Method Naming Convention | No | SetXX, GetXX are not the preferred method names as per our coding guidelines |
| Accessor Usage in Defining Class | Yes | Enabled |
| Accidental Concatenation | Yes | Enabled |
| Action Field Revalidated | No | not relevant |
| Action Method Returns Unknown Value | No | not relevant |
| Action Revalidated | No | not relevant |
| Add Method to Interface | Yes | Enabled |
| Allow compareTo to Throw Exceptions | Yes | Enabled |
| Always Override toString | No | Do we need this ? If needed we can use lombok to handle this |
| Always Use Identifier Names | No | not relevant |
| Annotation Type Naming Convention | Yes | Enabled |
| Anonymous Authentication in LDAP | No | not relevant |
| Anonymous Class in Loop | Yes | Enabled |
| Anonymous Class Member Visibility | Yes | Enabled |
| Append String | Yes | Enabled |
| Applet Field Security | No | not relevant |

| | | |
|---|---|---|
| Apply Dialog Font | No | not relevant |
| Array and non-Array Comparison | Yes | Enabled |
| Array Comparison | Yes | Enabled |
| Array Is Stored Without Copying | Yes | Enabled |
| Assignment In Condition | Yes | Enabled |
| Assignment to Non-final Static | Yes | Enabled |
| Attribute Injection | No | not relevant |
| Authentication By Hostname | No | not relevant |
| Avoid Accessing the FactoryBean Directly | No | not relevant |
| Avoid Auto-boxing | No | Should we enable this rule ? There are just too many instances where we are using autoboxing. |
| Avoid Auto-unboxing | No | Should we enable this rule ? There are just too many instances where we are using autoboxing. |
| Avoid Building Queries From User Input | No | not relevant |
| Avoid Class Loaders in EJB | No | not relevant |
| Avoid Comparing Classes By String Names | Yes | Enabled |
| Avoid Fields in Action Classes | No | not relevant |
| Avoid File IO in EJB | No | not relevant |
| Avoid Finalizers | Yes | Enabled |
| Avoid Future Keywords | Yes | Enabled |
| Avoid GUI in EJB | No | not relevant |
| Avoid Inner Classes | No | I think its fine to use Inner classes, Should we leave this diabled. |
| Avoid Instance Initializers | Yes | Enabled |
| Avoid Instantiation in Loops | Yes | Enabled |
| Avoid Instantiation to Get Class | Yes | Enabled |
| Avoid Loading Native Libraries | No | not relevant |
| Avoid Managing Threads | No | not relevant |
| Avoid Nested Assignments | No | There are number of places where we do things like " buf = readSizeBuf = new byte[4];" to make the code look concise. |
| Avoid Nested Blocks | Yes | Enabled |
| Avoid null Return Values | No | I think there are number of places where we return null. |
| Avoid Octal Literals | Yes | Enabled |
| Avoid Package Scope | No | This audit flags all inner classes, constructors, methods, and fields that have a package scope. Should we enable this ? |
| Avoid Passing this Reference as Argument | Yes | Enabled |

| | | |
|---|---|---|
| Avoid Primitive Method Parameters | No | I think this is against our coding style, primitives are preferred over warpper classes. |
| Avoid Similar Names | No | Not enabled. |
| Avoid Sockets in EJB | No | not relevant |
| Avoid StringBuffer Instantiation With Character Literal | Yes | Enabled |
| Avoid Subtyping Cloneable | Yes | Enabled |
| Avoid Synchronization in EJB | No | not relevant |
| Avoid the no-argument String constructor | Yes | Enabled |
| Avoid Unsafe Array Declaration | No | This audit rule flags an array declared public, static and final. - disabled. |
| Avoid Using "Field Access" Strategy | No | not relevant |
| Avoid Using "instanceof" | No | rule conflicts with other rules. |
| Avoid Using Autowiring | No | not relevant |
| Avoid Using Enterprise Schemas Version | No | not relevant |
| Avoid Utility Methods | No | Should we enable this rule ? |
| Badly Located Array Declarators | Yes | Enabled |
| Base64-Encoded Password | No | not relevant |
| Bean Members Should Be Serializable | No | not relevant |
| Beware of URL equals() and hashCode() | Yes | Enabled |
| Blank Line Usage | No | Not aligned to our coding standards. |
| Blank Password | No | not relevant |
| Block Depth | Yes | Enabled |
| Boolean Method Naming Convention | No | Not aligned to our coding standards. |
| Brace Position | No | taken care by code formatter. |
| Break with Label | Yes | Enabled |
| Bundle Activation Policy Compatibility (3.3) | Yes | Enabled |
| Business Logic in ActionForm | No | not relevant |
| Call Lock Without Unlock | Yes | Enabled |
| Caught Exceptions | Yes | Enabled |
| Character Comparison | Yes | Enabled |
| Check Enabled Before Logging | Yes | Enabled |
| Check Enabled Before Logging | Yes | Enabled |
| Check Type In Equals | Yes | Enabled |
| Class Extends java.security.Policy | No | not relevant |
| Class getName() Usage | Yes | Enabled |
| Class Naming Convention | Yes | Enabled |
| Class Should Be Final | No | This audit rule looks for classes that do not have any subclasses but are not marked as final. - Should we enable this ? |
| Class Should Define Validate Method | No | not relevant |

| | | |
|---|---|---|
| Class Should Have Final Static Fields | No | not relevant |
| Class Should Have Private Fields | No | not relevant |
| Class Should Validate All Fields | No | not relevant |
| Classes Should be Their Own Proxy | No | not relevant |
| Client Request Not Encrypted | No | not relevant |
| Client Request Not Signed | No | not relevant |
| Client Request Timestamp Not Signed | No | not relevant |
| Client Response Not Encrypted | No | not relevant |
| Client Response Not Signed | No | not relevant |
| Client Response Timestamp Not Signed | No | not relevant |
| Client Timestamp Does Not Expire | No | not relevant |
| Client Uses Username Token | No | not relevant |
| Clone Method Usage | Yes | Enabled |
| Clone Without Cloneable | Yes | Enabled |
| Cloneable Without Clone | Yes | Enabled |
| Close Connection Where Created | Yes | Enabled |
| Close Elements in Renderer | No | not relevant |
| Close In Finally | Yes | Enabled |
| Close Order | Yes | Enabled |
| Close Result Set Where Created | Yes | Enabled |
| Close Sessions Where Opened | No | not relevant |
| Close Statement Where Created | Yes | Enabled |
| Close Where Created | No | Creates lots flase positive - cases where resources need to be passed around or stored as instance variables we can't close these resources in the same block of code. |
| Code in Comments | No | Doesn't work correctly. |
| Code Injection | No | not relevant |
| Command Execution | Yes | Enabled |
| Command Injection | No | not relevant |
| Comment Local Variables | No | Should we enable this ? |
| Comparison Of Constants | Yes | Enabled |
| Comparison Of Incompatible Types | Yes | Enabled |
| Comparison Of Short And Char | Yes | Enabled |
| Complex Type Element Naming Convention | No | not relevant |
| Complex Type Naming Convention | No | not relevant |
| Concatenation In Appending Method | Yes | Enabled |
| Concurrent Modification | Yes | Enabled |
| Conditional Operator Use | No | Should we enable this ? We have used conditional operator in many places. |
| Configure Logging In File | Yes | |

| | | |
|---|---|---|
| Consistent Suffix Usage | Yes | Enabled |
| Constant Condition | Yes | Enabled |
| Constant Conditional Expression | Yes | Enabled |
| Constant Field Naming Convention | Yes | Enabled |
| Constants in Comparison | No | should we enable this ? |
| Constructors Only Invoke Final Methods | Yes | Enabled |
| Container Should Not Contain Itself As Element | Yes | Enabled |
| Continue with Label | Yes | Enabled |
| Convert Class to Interface | Yes | Enabled |
| Create Global Forward | No | not relevant |
| Cross-Site Scripting | No | not relevant |
| Cyclomatic Complexity | Yes | Enabled |
| Dangling Else | Yes | Enabled |
| Database Connections Should Not Be Static | Yes | Enabled |
| Date and Time Usage | Yes | Enabled |
| Debugging Code | Yes | Enabled |
| Declare Accessors for All ActionForm Fields | No | not relevant |
| Declare Accessors for Persistent Fields | No | not relevant |
| Declare As Interface | Yes | Enabled |
| Declare Default Constructors | No | Should we enable this rule? |
| Declare Identifier Properties | No | not relevant |
| Declare Private Identifier Setter | No | not relevant |
| Declare Setters for Bean Fields | No | not relevant |
| Declare Type for java.util.Date Property | No | not relevant |
| Declared Exceptions | Yes | Enabled |
| Default Namespace | No | not relevant |
| Default Not Last in Switch | Yes | Enabled |
| Define Constants in Interfaces | No | Should we enable this ? We have exceptions to this rule in our code. |
| Define Initial Capacity | No | Should we enable this ? We have exceptions to this rule in our code. |
| Define Load Factor | No | Should we enable this ? We have exceptions to this rule in our code. |
| Delete Temporary Files | Yes | Enabled |
| Denial of Service Threat | Yes | Enabled |
| Deploy Mappings With Mapped Classes | No | not relevant |
| Deprecated Method Found | Yes | Enabled |
| Dereferencing Null Pointer | Yes | Enabled |
| DeSerializeability Security | No | Rule enforces All classes to implement a readObject method - Not needed in my opinion |

| | | |
|---|---|---|
| Detect Multiple Iterations | Yes | Enabled |
| Disallow @Test Annotation | No | This rules states that @Test annotation should not be used. - Not sure why ? |
| Disallow Array Initializers | No | Arrays should not be statically initialized by an array initializer. Should we enable this rule ? |
| Disallow AST toString() | Yes | Enabled |
| Disallow Default Package | Yes | Enabled |
| Disallow Native Methods | Yes | Enabled |
| Disallow Notify Usage | Yes | Enabled |
| Disallow Sleep Inside While | Yes | Enabled |
| Disallow Sleep Usage | Yes | Enabled |
| Disallow Temporary Sessions | No | not relevant |
| Disallow ThreadGroup Usage | Yes | Enabled |
| Disallow Unnamed Thread Usage | Yes | Enabled |
| Disallow Use of AWT Peer Classes | Yes | Enabled |
| Disallow Use of Deprecated Thread Methods | Yes | Enabled |
| Disallow Yield Usage | Yes | Enabled |
| Disallowed File | Yes | Some files should not exist in a project. Should be enabled at a project level |
| Dispose Should Be Invoked | No | not relevant |
| Do not Access/Modify Security Configuration Objects | No | not relevant |
| Do not Catch IllegalMonitorStateException | Yes | Enabled |
| Do Not Create Finalizable Objects | No | Doesn’t allow to create objects like FileInputStream etc. |
| Do Not Declare Bindings | No | not relevant |
| Do Not Implement Outdated Interfaces | Yes | Enabled |
| Do Not Implement Serializable | No | This audit rule violates classes and interfaces that implement Serializable. Not releavant to us I think ? |
| Do Not Invoke setSize() | No | not relevant |
| Do Not Serialize Byte Arrays | No | This audit rule violates invocations of the method ObjectOutputStream.write(byte[]). - Not relevant to us I think. |
| Do Not Subclass ClassLoader | Yes | Enabled |
| Document Closing Braces | No | Should we enable this rule ? I think it creates clutter. |
| Don't Create Unused Error | No | not relevant |
| Don't Encode Markup in Renderer | No | not relevant |
| Don't Instantiate Beans | No | not relevant |

| | | |
|---|---|---|
| Don't Return Mutable Types | No | Don't return mutable types from methods. - should we enable this ? |
| Don't use concatenation to convert to String | Yes | Should we enable this rule ? We are using string concatenation in many places ? |
| Don't Use Default Bean Names | No | not relevant |
| Don't use HTML Comments | No | not relevant |
| Double Check Locking | Yes | Enabled |
| Duplicate Import Declarations | Yes | Enabled |
| Duplicate Property Name | Yes | Enabled |
| Duplicate Property Value | No | There can be many properties that have same value - e.g. transactional=true and locking=true, this rules flags them as errors. |
| Duplicate Validation Form | No | not relevant |
| Dynamic Dependency in Ivy | No | not relevant |
| Dynamic Dependency in Maven | Yes | Enabled |
| Dynamically Compose Test Suites | Yes | Enabled |
| Efficient Expression | Yes | Enabled |
| Either Nillable Or MinOccurs | No | not relevant |
| Empty Catch Clause | Yes | Enabled |
| Empty Class | Yes | Enabled |
| Empty Do Statement | Yes | Enabled |
| Empty Enhanced For Statement | Yes | Enabled |
| Empty Finalize Method | Yes | Enabled |
| Empty Finally Clause | Yes | Enabled |
| Empty For Statement | Yes | Enabled |
| Empty If Statement | Yes | Enabled |
| Empty Initializer | Yes | Enabled |
| Empty Method | Yes | Enabled |
| Empty Statement | Yes | Enabled |
| Empty String Detection | Yes | Enabled |
| Empty Switch Statement | Yes | Enabled |
| Empty Synchronized Statement | Yes | Enabled |
| Empty Try Statement | Yes | Enabled |
| Empty While Statement | Yes | Enabled |

| | | |
|---|---|---|
| Enforce Cloneable Usage | No | This audit rule requires almost All classes to override the clone method, Should we enable this ?<br>this rule flags non-anonymous classes that:<br>(1) do not implement Cloneable (so that the rule doesn't flag appropriate uses of Cloneable utilities),<br>(2) are non-final (final classes can't be extended),<br>(3) do not inherit a clone method (since inserting a clone method would be unnecessarily repetitive),<br>(4) and do not override clone(): |
| Enforce Singleton Property with Private Constructor | Yes | Enabled |
| Entity Bean's Remote Interface | No | not relevant |
| Entity Beans | No | not relevant |
| Entry Point Method | Yes | Enabled |
| Enumerated Type Naming Convention | Yes | Enabled |
| Enumeration Constant Naming Convention | Yes | Enabled |
| Environment Variable Access | No | Environment variables should not be accessed because not all platforms have support for environment variables. - We are using environment variable for setting project root path should we remove it ? |
| Equality Test with Boolean Literal | Yes | Enabled |
| Exception Creation | Yes | Enabled |
| Exception Declaration | No | This rule states that Exceptions should be declared to inherit from Exception, but not from either RuntimeException or RemoteException. Not aligned with our coding standards ? |
| Exception Parameter Naming Convention | Yes | Enabled |
| Explicit "this" Usage | Yes | Enabled |
| Explicit Invocation of Finalize | Yes | Enabled |
| Explicit Subclass of Object | Yes | Enabled |
| Expression Evaluation | Yes | Enabled |
| External Dependency in Ant | No | not relevant |
| External Dependency in Ivy | No | not relevant |

| | | |
|---|---|---|
| External Dependency in Maven | No | Probably it doesn't work correctly, flags "http://maven.apache.org/POM/4.0.0" to be 'external', |
| Extra Semicolon | Yes | Enabled |
| Fail Invoked in Catch | Yes | Enabled |
| Favor Static Member Classes over Non-Static | Yes | Enabled |
| Field Access Protection | Yes | Enabled |
| Field Javadoc Conventions | Yes | Enabled |
| Field Might Have Null Value | No | This is taken care by JSR 305, Should we enable this rule ? |
| Field Only Used in Inner Class | Yes | Enabled |
| File Comment | No | File comments are not enabled currently. |
| File Length | Yes | 2000 lines (all lines including comments.) |
| Filename Given Out | No | not relevant |
| Final Method Parameter In Interface | Yes | Enabled |
| Finalize Method Definition | Yes | Enabled |
| Finalize Should Not Be Public | Yes | Enabled |
| Float Comparison | Yes | Enabled |
| Floating Point Use | No | "Floating point values should rarely be used because of the potential for rounding errors." should we eanble this rule ? |
| Form Does Not Extend Validator Class | No | not relevant |
| Fully Parenthesize Expressions | No | Not needed I think, add unnecessary noise. |
| Getter and Setter Methods Should Be Final | No | Getter and setters are not as per our naming conventions, I think we can ingnore this ? |
| Handle Numeric Parsing Errors | Yes | Enabled |
| Hardcoded Password | Yes | Enabled |
| Hiding Inherited Fields | Yes | Enabled |
| Hiding Inherited Static Methods | Yes | Enabled |
| HTTP Response Splitting | No | not relevant |
| Illegal Main Method | Yes | Enabled |
| Implement a Zero-Argument Constructor | No | not relevant |
| Implement BeanNameAware Interface | No | not relevant |
| Implement Iterable | Yes | Enabled |
| Implicit Subclass of Object | No | it says ""Object" should be explicit specified as a superclass." Not needed I think ? |
| Import of Implicit Package | Yes | Enabled |
| Import Order | No | TBD |
| Import Style | Yes | Enabled |
| Improper calculation of array hashCode | Yes | Enabled |
| Improper conversion of Array to String | Yes | Enabled |
| Improper Use of Thread.interrupted() | Yes | Enabled |

| | | |
|---|---|---|
| Inappropriate Language | Yes | Enabled |
| Include Implementation Version | No | MF files are auto generated, "Manifest-Version:" is present but this rule still flags error. |
| Incompatible Renderer Type | No | not relevant |
| Incompatible types stored in a collection | Yes | Enabled |
| Incomplete reset() | No | not relevant |
| Incomplete State Storing | Yes | Enabled |
| Incomplete Validation Method | No | not relevant |
| Inconsistent Conversion Using toArray() | Yes | Enabled |
| Inconsistent Use of Override | Yes | Enabled |
| Inconsistent Validator Attribute | No | not relevant |
| Incorrect Argument Type | Yes | Enabled |
| Incorrect Use of equals() and compareTo() | Yes | Enabled |
| Indent Code Within Blocks | No | taken care by code formatter. |
| Index Arrays with Ints | Yes | Enabled |
| Inefficient use of toArray() | Yes | Enabled |
| Initialize Static Fields | Yes | Enabled |
| Instance Field Naming Convention | Yes | Enabled |
| Instance Field Security | No | disallows non-final public instance fields. - should we enable this rule ? (We'll need to make all the public/protected non-final fields as private and provide accessor methods) |
| Instance Field Visibility | Yes | Enabled |
| Integer Division in a Floating-point Expression | Yes | Enabled |
| Interface Naming Convention | Yes | Enabled |
| Invalid Check For Binding Equality | No | not relevant |
| Invalid Check For Java Model Identity | No | not relevant |
| Invalid DBC Tag Value | No | not relevant |
| Invalid Form Bean Class | No | not relevant |
| Invalid Loop Construction | Yes | Enabled |
| Invalid Property Type Mapping | No | not relevant |
| Invalid Source for Database Connection | No | not relevant |
| Invalid Visitor Usage | Yes | Enabled |
| Invocation of Default Constructor | Yes | Enabled |
| Invoke super.finalize() from within finalize() | Yes | Enabled |
| Invoke super.release() Within release() | No | not relevant |
| Invoke super.setUp() from within setUp() | Yes | Enabled |
| Invoke super.tearDown() from within tearDown() | Yes | Enabled |

| | | |
|---|---|---|
| Invoke super.validate() in validate() | No | not relevant |
| Invoke Synchronized Method In Loop | Yes | Enabled |
| JNDI Naming Standard | No | not relevant |
| JUnit Framework Checks | Yes | Enabled |
| Label Naming Convention | Yes | Enabled |
| Large Number of Constructors | Yes | Enabled |
| Large Number of Fields | Yes | Max 10 |
| Large Number of Methods | Yes | Max 15 |
| Large Number of Parameters | Yes | Max 7 |
| Large Number of Switch Statement Cases | Yes | Enabled |
| Lazily Initialize Singletons | Yes | Enabled |
| Line Length | No | taken care by code formatter. |
| Local Declarations | Yes | Variable declaration first in block - disabled.<br>Variable declaration first in method - disabled<br>Explicitly initialze each local variable - disabled<br>Declare each local variable in separate statement. - enabled |
| Local Variable Naming Convention | Yes | Enabled |
| Log Exceptions | Yes | also taken care by Log4e |
| Log Forging | Yes | Enabled |
| Log Level | Yes | Enabled |
| Loop Variable Naming Convention | Yes | Enabled |
| Loss of Precision in Cast | No | This rule doesn't allow to down cast from int to byte etc, there are many places in our code where we need to do this. |
| Manipulation with XPath | No | not relevant |
| Message Beans | No | not relevant |
| Message Document Must Have Version Attribute | No | not relevant |
| Method Chain Length | Yes | Enabled |
| Method Could Be Final | No | it says "Methods that are not supposed to be overridden should be declared final." - should we enabel this rule ? |
| Method Invocation in Loop Condition | No | It allows only some methods to be invoked inside loop conditions, there are number of exception to this rule in our code, should we enable this rule ? |
| Method Javadoc Conventions | Yes | Enabled |
| Method Naming Convention | Yes | Enabled |
| Method Parameter Naming Convention | Yes | Enabled |

| | | |
|---|---|---|
| Method Should Be Private | No | It says "Method should be private if it is not member of an interface and it doesn't override method of superclass and isn't overriden by subtype. " should we enable this rule ? |
| Method Should Be Static | No | "Methods that do not access any instance state or instance methods should be static." should we enable this rule ? |
| Mime Encoding Method Usage | No | not relevant |
| Minimize Scope of Local Variables | No | This rule recommends to convert the while-loop to a for-loop, for cases like 'while (beItr.hasNext())' - I think while-loop is looks cleaner in these cases. |
| Mismatched Notify | Yes | Enabled |
| Mismatched Wait | Yes | Enabled |
| Missing Application Context File | No | not relevant |
| Missing Assert in JUnit Test Method | Yes | Enabled |
| Missing Bean Description | No | not relevant |
| Missing Block | No | not consistent with our coding guidelines. I think ? |
| Missing Catch of Exception | No | not relevant |
| Missing Class in web.xml | No | not relevant |
| Missing Constants In Switch | Yes | Enabled |
| Missing Default in Switch | Yes | Enabled |
| Missing Error Page | No | not relevant |
| Missing Image File | No | not relevant |
| Missing Message | No | not relevant |
| Missing Message in Assert | Yes | Enabled |
| Missing Namespace Grouping Identifiers | No | not relevant |
| Missing Namespace Version | No | not relevant |
| Missing Or Misplaced Manifest Version | Yes | Enabled |
| Missing Or Misplaced Section Name | Yes | Enabled |
| Missing reset() Method | No | not relevant |
| Missing static method in non-instantiable class | Yes | Enabled |
| Missing Update in For Statement | Yes | Enabled |
| Missing Validator Name | No | not relevant |
| Misspelled Method Name | Yes | Enabled |
| Modifier Order | Yes | Enabled |
| More Than One Logger | Yes | Enabled |
| Multiple Return Statements | No | "Methods should have a single return statement." Should we enable this rule ? |

| | | |
|---|---|---|
| Multiplication Or Division By Powers of 2 | No | "Do not multiply or divide by powers of 2." The shift operator is faster and more efficient. Should we enable this rule ? |
| Mutability Of Arrays | No | "Do not return internal arrays from non-private methods." I don't think this is consistent with our thinking |
| Mutable Constant Field | No | "Disallows public static final mutable type fields." should we enable this rule ? |
| Nested Synchronized Calls | Yes | Enabled |
| Never Use the Identifier in equals() or hashCode() | No | not relevant |
| Next method invoked without hasNext method | Yes | Enabled |
| No Abstract Methods | Yes | Enabled |
| No Action For Validation | No | not relevant |
| No DB Driver Loading | No | not relevant |
| No Explicit Exit | Yes | Enabled |
| No Explicit This Use in EJB's | Yes | Enabled |
| No Instance Fields In Portlets | No | not relevant |
| No Instance Fields In Servlets | No | not relevant |
| No Public Members | Yes | Some classes can have only protected members e.g. classes that implement an abstract class. - This rule is enabled but some classes are excluded. |
| No Run Method | Yes | Enabled |
| No Set-up in Constructors | Yes | |
| No Such Field For Validation | No | not relevant |
| No Timestamp In Client Request | No | not relevant |
| No Timestamp In Client Response | No | not relevant |
| No Timestamp In Server Request | No | not relevant |
| No Timestamp In Server Response | No | not relevant |
| No Validation Message | No | not relevant |
| Non Static Logger | Yes | Enabled |
| Non-atomic File Operations | No | This rule flags things like (which is valid.)<br>if (databaseDir.exists())<br>throw new DBException("Database already exists"); |
| Non-blank Final Instance Field | Yes | Enabled |
| Non-case Label in Switch | Yes | Enabled |
| Non-conforming Backing Bean Methods | No | not relevant |
| Non-private Constructor in Static Type | Yes | Enabled |
| Non-protected Constructor in Abstract Type | Yes | Enabled |
| Non-serializable Class Declares readObject or | Yes | Enabled |

| | | |
|---|---|---|
| writeObject | | |
| Non-serializable Class Declares serialVersionUID | Yes | Enabled |
| Non-terminated Case Clause | Yes | Enabled |
| Nonce Not Used | No | not relevant |
| Notify method invoked while two locks are held | Yes | Enabled |
| Null Pointer Dereference | No | Doesn't seem to work as expected, JSR 305 works better. |
| Numeric Literals | No | ""Numeric literals should not appear in code." - should we enable this rule |
| Obey General Contract of Equals | No | "The equals method should compare the identity of the receiver and the argument, returning true if they are the same." 'identity' ? not sure what that means |
| Obsolete Modifier Usage | Yes | Enabled |
| One Class per Mapping File | No | not relevant |
| One Statement per Line | No | taken care by code formatter. |
| Overloaded Equals | Yes | Enabled |
| Overloaded Methods | No | "Overloading method names can cause confusion and errors." This audit rule finds methods that are overloaded. Overloaded methods are methods that have the same name and the same number of parameters, but do not have the same types of parameters.- Not consistent with our coding |
| Override both equals() and hashCode() | Yes | Enabled |
| Override Clone Judiciously | Yes | Enabled |
| Overriding a Non-abstract Method with an Abstract Method | Yes | Enabled |
| Overriding a Synchronized Method with a Non-synchronized Method | Yes | Enabled |
| Overriding Private Method | Yes | Enabled |
| Package Javadoc | No | TBD |
| Package Naming Convention | Yes | Enabled |
| Package Prefix Naming Convention | Yes | Enabled |
| Package Structure | Yes | All package names should start with 'com.paintedboxes' |
| Parenthesize Condition in Conditional Operator | Yes | Enabled |
| Parse Method Usage | No | "The parseXXX() methods for Numerics should not be used in an internationalized environment." should we enable this ? |
| Password in File | No | "Storing passwords in a configuration file is a security risk." should we enable this ? |

| | | |
|---|---|---|
| Path Manipulation | Yes | Enabled |
| Plain Text Password | Yes | Enabled |
| Platform Specific Line Separator | No | Doesn't seem to work as expected, reports error for things like '\t' , '\n' |
| Pluralize Collection Names | No | should we enable this ? |
| Possible Null Pointer | No | Taken care by JSR 305 annotations, which is a better way achieving this. |
| Potential Infinite Loop | Yes | Enabled |
| Pre-compute Constant Calculations | Yes | Enabled |
| Prefer Interfaces to Abstract Classes | No | This rule flags all the abstract classes. |
| Prefer Interfaces To Reflection | Yes | Enabled |
| Preferred Expression | Yes | Enabled |
| Prevent Overwriting Of Externalizable Objects | Yes | Enabled |
| Process Control | Yes | Enabled |
| Proper Finalize Usage | Yes | Enabled |
| Proper Servlet Usage | No | not relevant |
| Property File Must Exist | No | not relevant |
| Protected Member in Final Class | Yes | Enabled |
| Public Constructor in Non-public Type | Yes | Enabled |
| Public Nested Type in Package Visible Type | Yes | Enabled |
| Questionable Assignment | Yes | Enabled |
| Questionable Name | Yes | Enabled |
| Realm Debug Enabled | No | not relevant |
| Recursive Call With No Check | Yes | Enabled |
| Redirected With Password | No | not relevant |
| Redundant Assignment | Yes | Enabled |
| Referenced Class Not Defined | No | not relevant |
| Referenced Class Not Defined | No | not relevant |
| Reflection Injection | Yes | Enabled |
| Reflection Method Usage | No | "Don't use Class getMethod(), getField(), getDeclaredMethod() or getDeclaredField() methods in production code." should we enable this ? |
| Relative Access to Enterprise Schemas | No | not relevant |
| Relative Library Path | No | not relevant |
| Repeated Assignment | Yes | Enabled |
| Replace Synchronized Classes | Yes | Enabled |
| Request Message Naming Convention | No | not relevant |
| Request Parameters In Session | No | not relevant |
| Resource Injection | No | not relevant |
| Resource URL Manipulation | No | not relevant |

| | | |
|---|---|---|
| Response Message Naming Convention | No | not relevant |
| Restricted Packages | Yes | Enabled |
| Restricted Superclasses | Yes | Enabled |
| ReThrown Exceptions | Yes | Enabled |
| Return Boolean Expression Value | Yes | Enabled |
| Return Constant From getComponentType | No | not relevant |
| Return Constant From getFamily | No | not relevant |
| Return Constant From getRendererType | No | not relevant |
| Return in Finally | Yes | Enabled |
| Reusable Immutables | Yes | Enabled |
| Reuse DataSources for JDBC Connections | No | not relevant |
| Rollback Transaction on Exception | No | not relevant |
| Runtime Method Usage | Yes | Enabled |
| Same Validator Name | No | not relevant |
| Serializable Usage | Yes | Enabled |
| Serializeability Security | Yes | Enabled |
| Server Request Not Encrypted | No | not relevant |
| Server Request Not Signed | No | not relevant |
| Server Request Timestamp Not Signed | No | not relevant |
| Server Response Not Encrypted | No | not relevant |
| Server Response Not Signed | No | not relevant |
| Server Response Timestamp Not Signed | No | not relevant |
| Server Timestamp Does Not Expire | No | not relevant |
| Server Uses Username Token | No | not relevant |
| Service Naming Convention | No | not relevant |
| Service Operation Naming Convention | No | not relevant |
| Service Provider Request Security Not Configured | No | not relevant |
| Service Provider Response Security Not Configured | No | not relevant |
| Service Requester Request Security Not Configured | No | not relevant |
| Service Requester Response Security Not Configured | No | not relevant |
| Session Beans | No | not relevant |
| Session Must Be Synchronized | No | not relevant |
| Session-scope Form Needs reset() | No | not relevant |
| Simple Type Element Naming Convention | No | not relevant |
| Simple Type Naming Convention | No | not relevant |
| Sleep method invoked in synchronized code | Yes | Enabled |
| Sockets in Servlets | No | not relevant |
| Source Length | Yes | Enabled |
| Space After Casts | No | taken care by code formatter. |

| | | |
|---|---|---|
| Space After Commas | No | taken care by code formatter. |
| Space Around Operators | No | taken care by code formatter. |
| Space Around Periods | No | taken care by code formatter. |
| Specify an Error Page | No | not relevant |
| Specify Schema Version | No | not relevant |
| Specify SOAP Actions For WSDL Operations | No | not relevant |
| Spell Check Comments | Yes | Enabled |
| Spell Check Identifiers | Yes | Enabled |
| Spell Check Property Comments | Yes | Enabled |
| Spell Check Property Names | No | not relevant |
| Spell Check Property Values | No | not relevant |
| Spell Check String Literals | Yes | Enabled |
| Spell Check XML Attribute Names | Yes | Enabled |
| Spell Check XML Attribute Values | Yes | Enabled |
| Spell Check XML Body Text | Yes | Enabled |
| Spell Check XML Comments | Yes | Enabled |
| Spell Check XML Tag Names | Yes | Enabled |
| SQL Injection | Yes | Enabled |
| Start Method Invoked In Constructor | Yes | Enabled |
| Statement Creation | No | not relevant |
| Static Field Naming Convention | Yes | Enabled |
| Static Field Security | No | TBD |
| Static Instantiation | Yes | Enabled |
| Static Member Access | Yes | |
| String Comparison | Yes | Enabled |
| String Concatenation | No | should we enble this rule which doesn't allow String concatenation anywhere in the code? |
| String Concatenation in Loop | Yes | Enabled |
| String Created from Literal | Yes | Enabled |
| String indexOf Use | Yes | Enabled |
| String Literals | Yes | Enabled |
| String Method Usage | No | Disallows usage of String.equals(), String.equalsIgnoreCase() and String.comparesTo() |
| StringTokenizer Usage | No | Disallows usage of StringTokenizer |
| Struts Validator Not in Use | No | not relevant |
| Subclass should override method | yes | Enabled |
| Synchronization On getClass Method | Yes | Enabled |
| Synchronization On Non Final Fields | Yes | Enabled |
| Synchronized In Loop | Yes | Enabled |

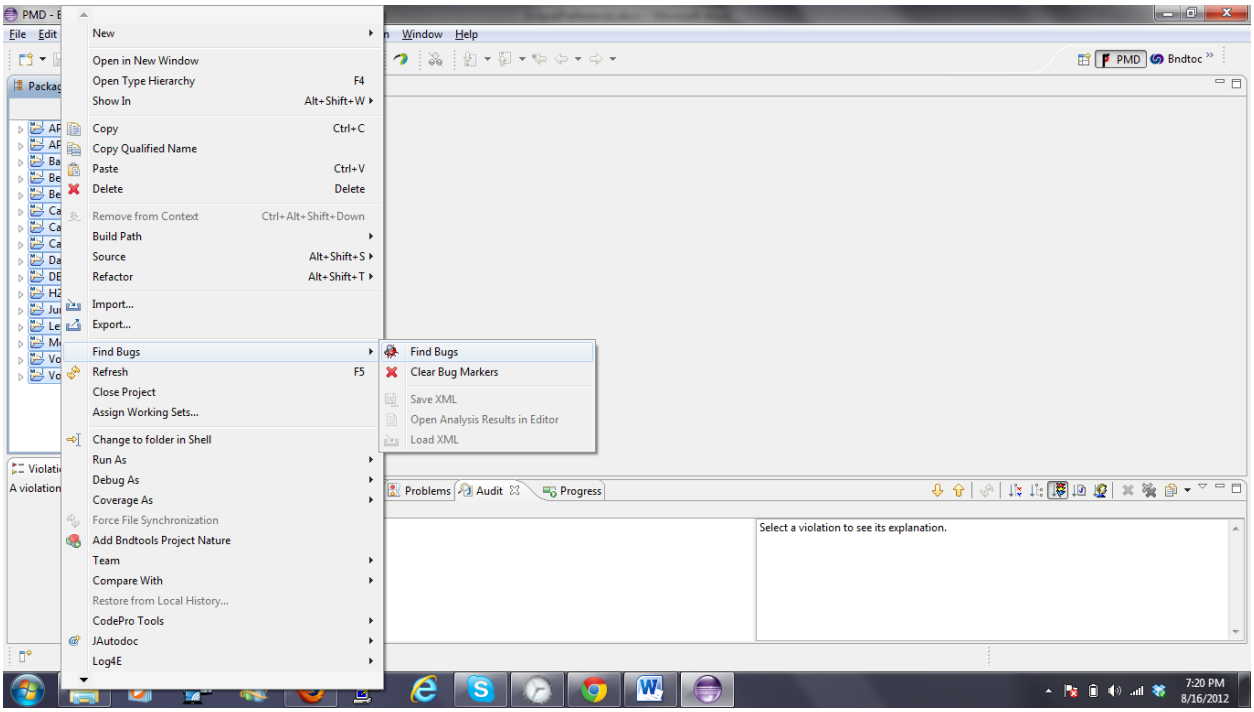
| | | |
|---|---|---|
| Synchronized Method | No | "Methods should never be marked as synchronized." |
| Tag Handler Field Not Cleared | No | not relevant |
| Tag Handler Should Implement Release | No | not relevant |
| Tainted Filter | No | not relevant |
| Tainted Internet Address | No | not relevant |
| Temporary Object Creation | Yes | Enabled |
| Test Case Naming Convention | No | TBD |
| Throw in Finally | Yes | Enabled |
| Thrown Exceptions | Yes | Enabled |
| toString() Method Usage | Yes | Enabled |
| Transient Field in Non-Serializable Class | Yes | Enabled |
| Type Declarations | Yes | Enabled |
| Type Depth | Yes | Enabled |
| Type Javadoc Conventions | Yes | Enabled |
| Type Member Ordering | No | Should we enable this ? Or we can have eclipse 'sort members' action. ? |
| Type Names Must Be Singular | No | Flags class names like Columns, Bytes. |
| Type Parameter Naming Convention | Yes | Enabled |
| Unassigned Field | No | reports too many false positives |
| Undefined Message Key | No | not relevant |
| Undefined Placeholder | No | not relevant |
| Undefined Property | Yes | Enabled |
| Unhashable class in hashed collection | Yes | Enabled |
| Unique Namespace per Schema | No | not relevant |
| Unknown Cast | Yes | Enabled |
| Unknown Component Type | No | not relevant |
| Unknown Renderer Type | No | not relevant |
| Unnecessary "instanceof" Test | Yes | Enabled |
| Unnecessary Catch Block | Yes | Enabled |
| Unnecessary Default Constructor | Yes | Enabled |
| Unnecessary Exceptions | No | If a method throws an exception which is a subclass of the exception mentioned in the throws clause, this rule flags such cases. |
| Unnecessary Final Method | Yes | Enabled |
| Unnecessary Import Declarations | Yes | Enabled |
| Unnecessary Null Check | Yes | Enabled |
| Unnecessary Override | Yes | Enabled |
| Unnecessary Return | Yes | Enabled |
| Unnecessary Return Statement Parentheses | No | Conflicts with other rules. |
| Unnecessary toString() Method Invocation | Yes | Enabled |

| | | |
|---|---|---|
| Unnecessary Type Cast | Yes | Enabled |
| Unregistered Validator | No | not relevant |
| Unsupported Clone | Yes | Enabled |
| Unused Assignment | No | Does't work correctly I think - I enabled this and got too many false violitions |
| Unused Field | Yes | Enabled |
| Unused Label | Yes | Enabled |
| Unused Method | Yes | Enabled |
| Unused Return Value | No | There are many cases where we don't want to use return value e.g. map.put() etc. should we enable this rule ? |
| Unused StringBuffer | yes | Enabled |
| Unused StringBuilder | Yes | Enabled |
| Unused Validation Form | No | not relevant |
| Unvalidated Action Field | No | not relevant |
| Unvalidated Action Form | No | not relevant |
| Unvalidated Action Form Field | No | not relevant |
| Usage Of Binary Comparison | Yes | Enabled |
| Usage Of Static Calendar | Yes | Enabled |
| Usage Of Static Date Format | Yes | Enabled |
| Use "for" Loops Instead of "while" Loops | No | rule checks for the use of "while" loops rather than "for" loops, I think while loops are cleaner in some cases |
| Use "Nullable" Type for Identifier Properties | No | not relevant |
| Use == to Compare With null | Yes | Enabled |
| Use @After Annotation Rather than tearDown() | Yes | Enabled |
| Use @Before Annotation Rather than setUp() | Yes | Enabled |
| Use @RunWith and @SuiteClasses to build test suite | Yes | Enabled |
| Use @Test Annotation for JUnit test | Yes | Enabled |
| Use a Valid Schema Namespace | No | not relevant |
| Use a Valid WSDL Namespace | No | not relevant |
| Use Action Interface | No | not relevant |
| Use Additional Attribute in <constructor-arg> Tag | No | not relevant |
| Use ApplicationContext to Assemble Beans | No | not relevant |
| Use arraycopy() Rather Than a Loop | Yes | Enabled |
| Use Available Constants | Yes | Enabled |
| Use Buffered IO | Yes | Enabled |
| Use Camel Case in Namespaces | No | not relevant |
| Use char Rather Than String | Yes | Enabled |
| Use charAt() Rather Than startsWith() | Yes | Enabled |

| | | |
|---|---|---|
| Use Compound Assignment | Yes | Enabled |
| Use deep Arrays methods when necessary | Yes | Enabled |
| Use Domain-specific Terminology | No | not relevant |
| Use equals() Rather Than == | Yes | Enabled |
| Use equals() Rather Than equalsIgnoreCase() | No | We need equalsIgnoreCase() in some cases. |
| Use Existing Global Forwards | Yes | Enabled |
| Use ForwardAction Where Possible | No | not relevant |
| Use html:messages Instead of html:errors | No | not relevant |
| Use idref Element | No | not relevant |
| Use Interceptor to Track Duplicate Submits | No | not relevant |
| Use Interfaces for Collection Attributes | No | not relevant |
| Use Interfaces Only to Define Types | No | not relevant |
| Use locale-specific methods | No | not relevant |
| Use NoSuchElementException in next() | No | not relevant |
| Use of "instanceof" with "this" | Yes | Enabled |
| Use Of Broken Or Risky Cryptographic Algorithm | No | not relevant |
| Use of instanceof in Catch Block | Yes | Enabled |
| Use of Random | No | Random is used only inside unit testing code. |
| Use of xs:any | No | not relevant |
| Use Only Non-Final Persistent Classes | No | not relevant |
| Use Only Static Inner Classes | No | not relevant |
| Use Or to Combine SWT Style Bits | No | not relevant |
| Use Privileged Code Sparingly | No | not relevant |
| Use Qualified Attributes | No | not relevant |
| Use Session-per-request Pattern | No | not relevant |
| Use Setter Injection | No | not relevant |
| Use Start Rather Than Run | Yes | Enabled |
| Use StringBuffer length() | Yes | Enabled |
| Use StrutsTestCase for Unit Testing | No | not relevant |
| Use SuccessAction Where Possible | No | not relevant |
| Use Thread-safe Lazy Initialization | Yes | Enabled |
| Use Type for Constructor Argument Matching | No | not relevant |
| Use Type-Specific Names | No | Mandates InputStream objects to have name like "in" |
| Use Valid SWT Styles | No | not relevant |
| Use Valid Type for Form-property | No | not relevant |
| Use valueOf() to wrap primitives | Yes | Enabled |
| Use WSDL First approach | No | not relevant |
| Use XmlBeanFactory | No | not relevant |
| Validate XML | Yes | Enabled |

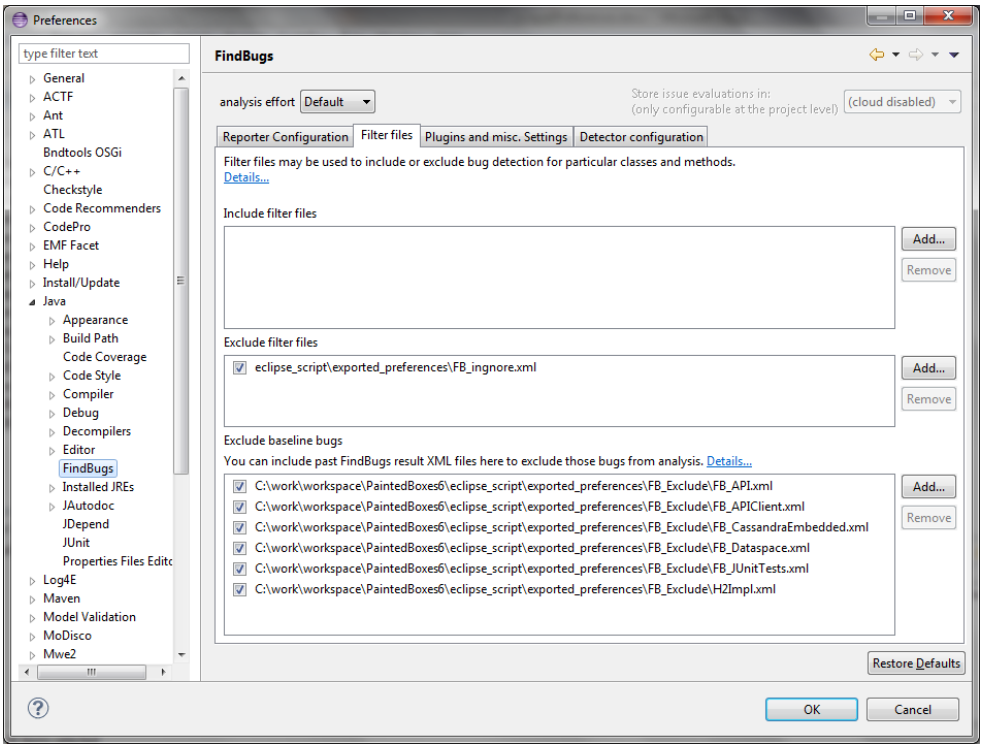
| | | |
|---|---|---|
| Validation Method Naming Convention | No | not relevant |
| Validation Not Enabled | No | not relevant |
| Validator Configuration File Does Not Exist | No | not relevant |
| Validator Disabled | No | not relevant |
| Variable Declared Within a Loop | No | TBD |
| Variable Has Null Value | No | Taken care by JSR 305 annotations, which is a better way achieving this. |
| Variable Should Be Final | Yes | Enabled |
| Variable Usage | Yes | Enabled |
| Wait Inside While | Yes | Enabled |
| Wait method invoked while two locks are held | Yes | Enabled |
| wait() Invoked Instead of await() | Yes | Enabled |
| Weblogic Session ID Length | No | not relevant |
| White Space Before Property Name | Yes | Enabled |
| White Space Usage | No | taken care by code formatter. |
| Wrong Family Returned | No | not relevant |
| Wrong Integer Type Suffix | Yes | Enabled |
| WSDL Must Specify ESB Endpoints | No | not relevant |
| WSDL Namespace for Included Schemas | No | not relevant |
| XSD File Naming Convention | No | not relevant |

## Find Bugs



## Note: Importing Findbugs Rules are imported along with Eclipse Preferences.

To exclude “false positive” errors we need to include files present in <Workspace>\ eclipse_script\exported_preferences\FB_Exclude




## List of Findbugs rules that are **Disabled**:

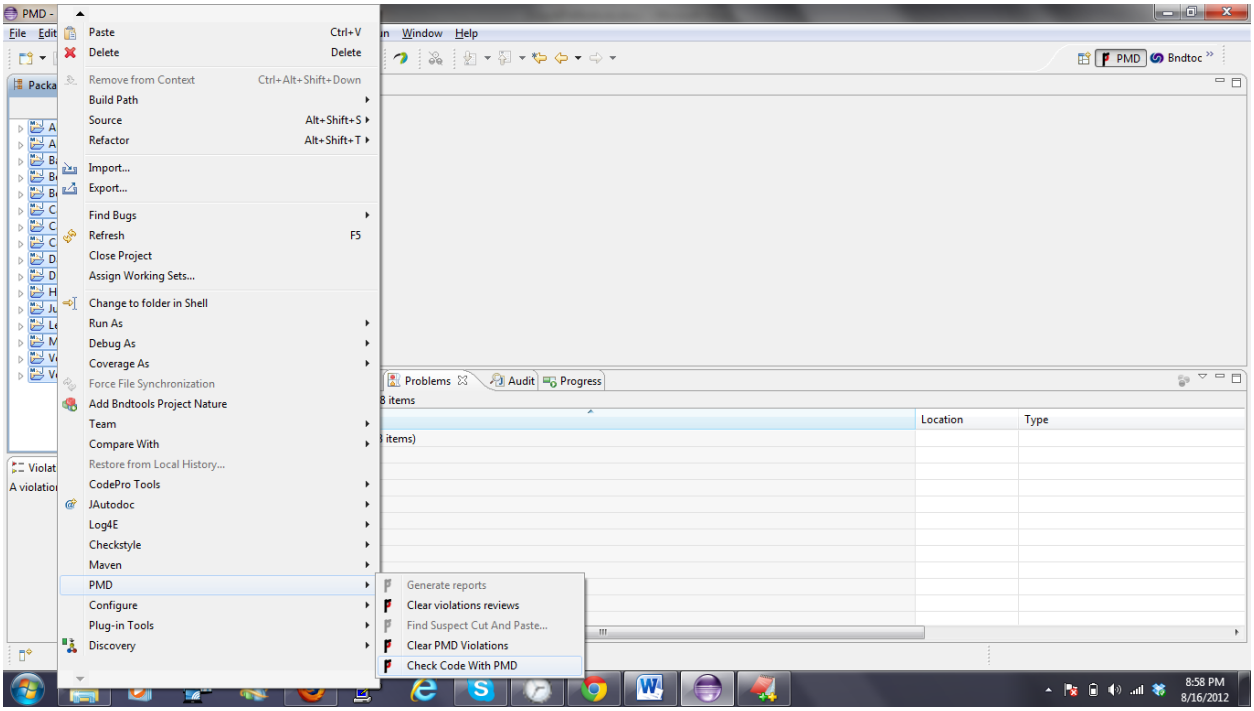
edu.umd.cs.findbugs.detect.UnreadFields - This detector looks for fields whose value is never read, Disabled this rules because it flags unread public fields for public interfaces.

edu.umd.cs.findbugs.detect.ReadReturnShouldBeChecked - This detector looks for calls to InputStream.read() or InputStream.skip() where the return value is ignored. – Should we enable this rule there are many places where we are not checking/using the return values, e.g. map.put(), remove() etc.

edu.umd.cs.findbugs.detect.FindCircularDependencies - This detector looks circular dependencies among classes. This rule is currently disabled as our base interface design has circular dependency (BETable uses BEDatabase and BEDatabase encapsulates BETable.)

## PMD

Running PMD rules




Note: PMD rules where imported into CodePro, most of the where successfully mapped to corresponding Codepro rules, the tool failed to convert **following rules**, Hence Only these rules were configured inside PMD

- AbstractClassWithoutAnyMethod
- AbstractNaming – **Disabled** currently
- AccessorClassGeneration
- AtLeastOneConstructor – **Disabled** currently
- AvoidDecimalLiteralsInBigDecimalConstructor
- AvoidDuplicateLiterals – **Disabled** currently, this flags cases like String literals mentioned inside logDebug statement if same literal is repeated.
- AvoidFieldNameMatchingMethodName – Disabled currently we are using method names same as field names for accessors/mutators.
- AvoidFieldNameMatchingTypeName
- AvoidFinalLocalVariable – Disabled, Conflicts with other rules which say that if local variable is assingned only once then make it final.
- AvoidMultipleUnaryOperators - Enabled
- AvoidRethrowingException - Enabled
- AvoidStringBufferField - Enabled
- AvoidThrowingNewInstanceOfSameException - Enabled
- AvoidUsingHardCodedIP - Enabled
- AvoidUsingShortType – “Java uses the 'short' type to reduce memory usage, not to optimize calculation. In fact, the jvm does not have any arithmetic capabilities for the short type: the jvm must convert the short into an int, do the proper calculation and convert the int back to a short. So, the use of the 'short' type may have a greater impact than memory usage.” Should we NOT use short?
- AvoidUsingVolatile - Enabled
- BeanMembersShouldSerialize - If a class is a bean, or is referenced by a bean directly or indirectly it needs to be serializable. Member variables need to be marked as transient, static, or have accessor methods in the class. Marking

variables as transient is the safest and easiest modification. Accessor methods should follow the Java naming conventions, i.e.if you have a variable foo, you should provide getFoo and setFoo methods. Flags almost all non-static fields and it cannot detect accessor methods because we don't use Getter method naming convention.

- BooleanInversion - Enabled
- CallSuperInConstructor - Enabled
- CheckResultSet - Enabled
- ClassWithOnlyPrivateConstructorsShouldBeFinal - Enabled
- CloneMethodMustImplementCloneable - Enabled
- CollapsibleIfStatements - Enabled
- ConfusingTernary – Disabled – “In an "if" expression with an "else" clause, avoid negation in the test.” This flags statements like “if(obj != null){ ..} else {..}, should we enable this rule and refactor our code ?
- ConsecutiveLiteralAppends - Enabled
- CouplingBetweenObjects - Enabled
- DataflowAnomalyAnalysis - Enabled
- EmptyInitializer - Enabled
- EmptyMethodInAbstractClassShouldBeAbstract – Disabled - Rule doesn't make sense since we need empty implementations to some abstract methods.
- ExceptionAsFlowControl - Enabled
- ExcessiveImports - Enabled
- ExcessivePublicCount - Enabled
- FinalizeOnlyCallsSuperFinalize - Enabled
- ForLoopShouldBeWhileLoop - Enabled
- InefficientEmptyStringCheck - Enabled
- JUnitUseExpected - Enabled
- JumbledIncrementer - Enabled
- LocalHomeNamingConvention - Enabled
- LocalInterfaceSessionNamingConvention - Enabled
- NPathComplexity - Enabled
- NcssConstructorCount - Enabled
- NcssMethodCount - Enabled
- NcssTypeCount - Enabled
- NullAssignment - Enabled
- PositionLiteralsFirstInComparisons - Enabled
- PreserveStackTrace - Enabled
- RemoteInterfaceNamingConvention - Enabled
- RemoteSessionInterfaceNamingConvention - Enabled
- ShortMethodName - Enabled
- SimpleDateFormatNeedsLocale - Enabled
- SimplifyBooleanAssertion - Enabled
- SimplifyConditional - Enabled
- SingularField - Enabled
- SuspiciousConstantFieldName – Disabled - Not compatible with our coding guidelines, we use all caps for static members, this rule suspects them to be ‘constants’
- SuspiciousOctalEscape - Enabled
- SwitchDensity - Enabled
- TestClassWithoutTestCases - Enabled
- TooFewBranchesForASwitchStatement - Enabled
- TooManyStaticImports – Disabled currently, static imports are configured to be auto generated by eclipse ‘Save Action”

- UnnecessaryBooleanAssertion - Enabled
- UnnecessaryCaseChange - Enabled
- UnnecessaryLocalBeforeReturn - Enabled
- UnsynchronizedStaticDateFormatter - Enabled
- UnusedNullCheckInEquals - Enabled
- UseArraysAsList - Enabled
- UseAssertEqualsInsteadOfAssertTrue - Enabled
- UseAssertNullInsteadOfAssertTrue - Enabled
- UseAssertSameInsteadOfAssertTrue - Enabled
- UseCollectionIsEmpty - Enabled
- UseCorrectExceptionLogging - Enabled
- UseIndexOfChar - Enabled
- UseProperClassLoader - Enabled
- UseSingleton – Disabled same rule is available in Checkstyle.
- UselessOperationOnImmutable - Enabled
- UselessStringValueOf - Enabled

# Appendix

## CodePro Rule references:

### 'ANYONE' Method Permission

**Severity:** Medium

**Summary**

Method permission that grants access to the 'ANYONE' role should not be used.

**Description**

Method permission that grants access to the 'ANYONE' role usually indicates a loose system access design that was not thought through thoroughly enough.

**Security Implications**

Such a weak design can produce security holes that can be used by an attacker to access secured data.

**Example**

The following declaration would be flagged as a violation because it sets the 'ANYONE' role to a method access permission:

```
<ejb-jar>
<assembly-descriptor>
<method-permission>
<role-name>ANYONE</role-name>
<method>
<ejb-name>Account</ejb-name>
<method-name>getBalance</method-name>
</method>
</method-permission>
</assembly-descriptor>
</ejb-jar>
```

### Abstract Specialization

**Severity:** Medium

**Summary**

Abstract classes should not be subclasses of concrete classes.

**Description**

This audit rule finds abstract classes that are subclasses of concrete classes. An exception is made for abstract subclasses of the class java.lang.Object.

**Example**

The following class declaration would be flagged as a violation because the class java.util.ArrayList is not an abstract class:

```
public abstract class SpecializedList extends ArrayList
```

while the following class declaration would not be flagged because the class java.lang.Object is treated specially:

```
public abstract Person extends Object
```

**Accessor Method Naming Convention**

**Severity:** Medium

**Summary**

The prefixes "get" and "set" should only be used for accessor methods.

**Description**

This audit rule checks for methods whose names begin with either "get" or "set" that are not accessor methods. A "getter" can have any number of parameters, but it must return a value. A "setter" must have at least one parameter and must not return a value.

**Example**

The following method would be flagged as a violation because it begins with the word "get", but does not return a value:

```
public void getUserName()
{
```

```
...
}
```

**Accessor Usage in Defining Class**

**Severity:** Medium

**Summary**

Fields should be referenced directly in their declaring type.

**Description**

This audit rule finds places in types that declare fields where a declared field could be directly referenced but is instead being indirectly referenced through an accessor method. Such indirect reference is unnecessary and should therefore be avoided.

**Example**

Given a class that declares the following field and accessor method:

```
private int itemCount = 0;

public int getItemCount()
{
return itemCount;
}
```

The following usage of the accessor method would be flagged as a violation if it occurred within the class:

```
if (getItemCount() > 0) ...
```

**Accidental Concatenation**

**Severity:** Medium

**Summary**

Two numbers concatenated without any characters in between is probably an error.

**Description**

This audit rule finds places where two or more numbers are being concatenated without intervening strings or characters. This is usually a mistake caused by forgetting to parenthesize the sub-expression.

**Example**

The addition of the two integers in the code below would be flagged as a violation:

```
public String getSummary(int passCount, int failCount) {
return "Of the " + passCount + failCount + " students, "
+ passCount + " passed and " + failCount + " failed.";
}
```

**Action Field Revalidated**

**Severity:** Medium

**Summary**

Only one `<field-validator>` should be declared for one action field.

**Description**

This audit rule violates `<field-validator>` declarations that repeat more than once for one action field.

**Security Implications**

When action field validator is declared more than once for one action field this could lead to an unexpected behavior or failure to validate the user input properly that could possibly allow an attacker to perform some kind of remote attack on a system.

**Example**

The following declaration in `MyAction-validation.xml` declares validator for the field `name` twice and would thus be marked as violation:it results in an found bind:

```
<field name="name">
<field-validator type="required">
<message>Name is required.</message>
</field-validator>
<field-validator type="required">
```

```
<message>Please enter your name.</message>
</field-validator>
</field>
```

**Action Method Returns Unknown Value**

**Severity:** Medium

**Summary**

Action methods should return values declared in the configuration file.

**Description**

This audit rule looks for action methods in backing beans that return a constant outcome value that is not declared in faces-config.xml. Because the JSF configuration file defines navigation rules that map outcomes into pages, there is no point in returning an outcome which is never referred to in faces-config.xml.

**Example**

A violation will be signaled if "checkout" outcome is never mentioned in the configuration file.

```
class MyPageBean {
...
public String proceedToCheckout() {
return "checkout";
}
...
}
```

**Action Revalidated**

**Severity:** Medium

**Summary**

Only one `ActionName-validation.xml` file should be declared for one action.

**Description**

This audit rule violates `ActionName-validation.xml` declarations that are found more than once in the project.

**Security Implications**

When an action validator is declared more than once for one action, this could lead to an unexpected behavior or failure to validate the user input properly which could possibly allow an attacker to perform some kind of remote attack on a system.

**Example**

A validation file `MyAction-validation.xml` for MyAction that is present in more than one place in the project will be marked as a violation.

**Add Method to Interface**

**Severity:** Medium

**Summary**

Identify properties that can be added to an interface.

**Description**

This audit rule finds methods in concrete classes that can be added to their corresponding interfaces. If a class is named Foo and has a "bar" property with getBar() and setBar() methods, the corresponding interface, IFoo, will be checked to see if it defines the getter and setter method.

**Example**

If the class Employee were defined as:

```
public class Employee implements IEmployee
```

and defined the methods getSSN() and setSSN() that were not declared in the IEmployee interface, then those methods would be flagged as violations.

**Allow compareTo to Throw Exceptions**

**Severity:** Medium

**Summary**

The compareTo method is expected to throw ClassCastException and NullPointerException.

**Description**

It is not necessary to test the value of the argument to compareTo prior to casting it. If the argument is null a NullPointerException should be thrown. If it is of the wrong type a ClassCastException should be thrown. This rule finds implementations of the compareTo method that explicitly catch those exceptions when they should not.

**Example**

The following implementation of compareTo tests the argument against null and uses the instanceof operator to check the type of the argument:

```
public int compareTo(Object o)
{
if (o == null) {
return 1;
} else if (!(o instanceof MyClass)) {
return 0;
}
return value - ((MyClass) o).value;
}
```

It should be replaced by the following, cleaner, implementation:

```
public int compareTo(Object o)
{
return value - ((MyClass) o).value;
}
```

**Always Override toString**

**Severity:** Medium

**Summary**

Every class should override toString().

**Description**

This audit rule identifies non-abstract classes that do not override the toString method. It ignores classes that cannot be instantiated, and can be configured to ignore other common situations, such as classes that inherit an implementation from a superclass other than Object.

**Example**

The following class would be flagged as needing an implementation of toString():

```
public class StringHolder
{
private String value;

public StringHolder(String initialValue)
{
value = initialValue;
}

public String getValue()
{
return value;
}
}
```

**Always Use Identifier Names**

**Severity:** Medium

**Summary**

You should always specify a name attribute for identifiers in Hibernate.

**Description**

This audit rule looks for identifier declarations that do not include a name attribute. If you do not specify a name attribute you allow Hibernate to manage database identity internally. For example, consider the following mapping declaration:

```
<id column="CATEGORY_ID">
<generator class="native"/>
</id>
```

Hibernate will now manage the identifier values internally. But this technique has a serious drawback: you can no longer use Hibernate to manipulate detached objects effectively. So, you should always specify a name attribute in Hibernate. (If you don't like them being visible to the rest of your application, make the accessor methods private.)

**Example**

The following identifier declaration would be flagged because it does not include a name attribute:

```
<id column="CATEGORY_ID">
<generator class="native"/>
</id>
```

you should use something like following:

```
<id name="category" column="CATEGORY_ID">
<generator class="native"/>
</id>
```

**Annotation Type Naming Convention**

**Severity:** Medium

**Summary**

Annotation types names should conform to the defined standard.

**Description**

This audit rule checks the names of all annotation types.

**Example**

If the rule were configured to require that all annotation type names start with a capital letter, the following declaration would be flagged as a violation:

```
public @interface internalUseOnly ...
```

**Anonymous Authentication in LDAP**

**Severity:** Medium

**Summary**

Enforces authorized access to LDAP.

**Description**

This audit rule finds places where the `Context.SECURITY_AUTHENTICATION` property is set to `"none"`, resulting in an anonymous binding.

**Security Implications**

When LDAP data is accessible with anonymous authentication, a malicious user can have access to information that would otherwise be inaccessible.

**Example**

The following method invocation would be flagged as a violation because it results in an anonymous bind:

```
env.put(Context.SECURITY_AUTHENTICATION, "none");
```

**Anonymous Class in Loop**

**Severity:** Medium

**Summary**

Placing an anonymous class inside a loop can decrease performance.

**Description**

Placing the creation of an instance of an anonymous class inside a loop will create a new instance each time the loop body is executed. Because the state of the object cannot depend on the state of any variables that change within the loop, a single object can usually be created outside the loop, increasing performance.

**Example**

In the following code:

```
EventGenerator[] generators;

for (int i = 0; i < generators.length; i++) {
generators[i].addListener(new Listener() {
public void eventGenerated()
{
...
}
```

```
}
}
```

The creation of the listener could, and should, be moved outside the loop because a single listener could easily be shared by all of the event generators.

**Anonymous Class Member Visibility**

**Severity:** Medium

**Summary**

Members of anonymous classes should be private.

**Description**

This audit rule checks the visibility of all fields and methods within an anonymous class to ensure that they are declared private (except for methods that override an inherited method). Such fields and methods should be declared private to make it clear that they cannot be accessed outside the anonymous class (and to make it easier to detect dead code).

**Example**

The following field declaration would be flagged as a violation because it is declared as being public:

```
new Runnable() {
public int x;
...
}
```

**Append String**

**Severity:** Medium

**Summary**

Appending strings with single characters to buffers or streams is slower than appending just the single character.

**Description**

This audit rule finds single character string literals as a single argument to a method invocation where that argument can be replaced by a character literal to improve performance.

**Example**

Given the following declaration:

```
StringBuffer sb = new StringBuffer();
```

The statement

```
sb.append("a");
```

would be flagged as needing to be replaced by the statement

```
sb.append('a');
```

**Applet Field Security**

**Severity:** Medium

**Summary**

Enforces Applet fields to be non-private, final and non-static

**Description**

This audit rule violates all field declarations in Applets that are not private, final and static.

**Security Implications**

Fields that have all of these characteristics reduce the risk of malicious users from manipulating or gaining internal access to the Applet.

**Example**

The following integer would be flagged as it is public:

```
public final int x = 0;
```

**Apply Dialog Font**

**Severity:** Medium

**Summary**

The method Dialog.applyDialogFont(Composite) should be invoked in the createContents method so that user preferences will be honored.

**Description**

This audit rule finds implementations of the method createContents(Composite) in subclasses of org.eclipse.jface.dialogs.Dialog in which the method Dialog.applyDialogFont(Composite) is not invoked. This method should always be invoked so that user preference settings will be honored.

**Example**

The following method:

```
protected Control createContents(Composite parent)
{
Composite contents;

contents = (Composite) super.createContents(parent);
...
return contents;
}
```

Should have the following line added just before the return statement:

```
Dialog.applyDialogFont(parent);
```

**Array and non-Array Comparison**

**Severity:** Medium

**Summary**

Avoid using the `equals()` method to compare array and non-array types because this call always returns false.

**Description**

This rule looks for places where the `equals()` method is used to compare array and non-array types. Such a comparison will always return `false`, and thus usually indicates an error.

**Example**

The following code whould be flagged as a violation because it contains an incorrect comparison:

```
public void myMethod(String a[]) {
if(a.equals("INCORRECT"))
return;
}
}
```

**Array Comparison**

**Severity:** Medium

**Summary**

Arrays should not be compared using equals (==), not equals (!=), or equals().

**Description**

Arrays should always be compared using one of the comparison methods defined for arrays. This audit rule looks for comparisons using either the equals (==) or not equals (!=) operators or the equals() method.

**Example**

```
char[] currentName, proposedName;

...
if (proposedName != currentName) {
...
}
```

**Array Is Stored Without Copying**

**Severity:** Medium

**Summary**

Storing of arrays without copying should not be used.

**Description**

This audit rule looks for places where arrays are stored without copying.

**Security Implications**

If constructors and methods receive and store arrays without copying, these arrays could be unpredictably changed from outside of the class.

**Example**

The following declaration of the `setArray` method will be marked as a violation because it does not copy its parameter:

```
private String[] array;
....
public void setArray( String[] newArray){
this.array = newArray;
}
```

**Assignment In Condition**

**Severity:** Medium

**Summary**

The assignment operator should never be used in a condition.

**Description**

This audit rule finds places in the code where an assignment operator is used within a condition associated with an if, for, while or do statement. Such uses are often caused by mistyping a single equal (=) where a double equal (==) was intended.

**Example**

```
if (a = 0) {
...
}
```

**Assignment to Non-final Static**

**Severity:** Low

**Summary**

Static fields should only be changed in static methods.

**Description**

Assignments to a static field in a non-static context are usually not intended, and therefore usually represent an error.

**Example**

```
public class Foo {
static int x = 2;
public doSomething(int y) {
x = y;
}
}
```

**Attribute Injection**

**Severity:** High

**Summary**

Request parameters and other tainted data should not be passed into the `BasicAttribute` of a LDAP context search request without sanitizing.

**Description**

This audit rule violates all `BasicAttribute` instance creations where tainted data is used.

**Security Implications**

Only trusted data should be passed to a search request as an attribute, otherwise an attacker could possibly create a request that will provide access to otherwise inaccessible and sensitive data.

**Example**

The following code creates a `BasicAttribute` with the use of unsanitized data retrieved from `HttpServletRequest` parameter and does not clean this data:

```
LdapContext ctx = getLdapContext();
String userName = req.getParameter("user");
```

```
BasicAttribute attr = new BasicAttribute("userName", userName);
try {
NamingEnumeration users = ctx.search("ou=People,dc=example,dc=com", attr,
null);
```

**Authentication By Hostname**

**Severity:** Medium

**Summary**

Do not use hostnames to authenticate users.

**Description**

This rule looks for places where the method `getCanonicalHostName()` is used in a condition.

**Security Implications**

When security decisions are made based on trusting certain hostnames, DNS cache poisoning attack can be used to access secured parts of an application.

**Example**

The following check would be flagged as a violation because the decision is made based on a host name:

```
if (addr.getCanonicalHostName().endsWith("trustme.com")) {
trusted = true;
}
```

**Avoid Accessing the FactoryBean Directly**

**Severity:** Medium

**Summary**

Avoid accessing the FactoryBean directly and invoking getObject() manually.

**Description**

This audit rule looks for places where a FactoryBean is accessed directly. Accessing the FactoryBean directly is actually very simple: you simply prefix the bean name with an ampersand in the call to getBean(). This feature is used in a few places in the Spring code, but your application should really have no reason to use it. The FactoryBean interface is intended to be used as a piece of supporting infrastructure to allow you to use more of your application's classes in an IoC setting. Avoid accessing the FactoryBean directly and invoking getObject() manually; if you do not, you are making extra work for yourself and are unnecessarily coupling your application to a specific implementation detail that could quite easily change in the future.

**Example**

The invocation factory.getBean("&shaDigest") in the following code would be flagged because it uses direct access to a FactoryBean:

```
public class AccessingFactoryBeans {

public static void main(String[] args) {
BeanFactory factory = new XmlBeanFactory(new FileSystemResource("..."));

MessageDigest digest = (MessageDigest) factory.getBean("shaDigest");

MessageDigestFactoryBean factoryBean =
(MessageDigestFactoryBean) factory.getBean("&shaDigest");
}
}
```

**Avoid Auto-boxing**

**Severity:** Medium

**Summary**

The auto-boxing of primitive values should be avoided.

**Description**

This audit rule looks for places where auto-boxing is used to create a wrapper object for a primitive value. The problem with allowing the compiler to generate this code is that it is easy to miss the fact that it is happening.

**Example**

The following code would be flagged as a violation because the primitive value on the right must be wrapped in an instance of Integer:

```
Integer value = 3;
```

**Avoid Auto-unboxing**

**Severity:** Medium

**Summary**

The auto-unboxing of primitive value wrappers should be avoided.

**Description**

This audit rule looks for places where auto-unboxing is used to access a primitive value from a wrapper object. The problem with allowing the compiler to generate this code is that it is easy to miss the fact that it is happening.

**Example**

The following code would be flagged as a violation because the wrapper object on the right must be dereferenced in order to access the primitive value:

```
int value = new Integer(3);
```

**Avoid Building Queries From User Input**

**Severity:** High

**Summary**

Database queries should not be constructed from user input.

**Description**

This audit rule looks for places where a query is being performed that was constructed from data that might have come from user input.

Specifically, this audit rule violates the usage of the methods "execute", "executeQuery", or "executeUpdate" as defined in `java.sql.Statement`.

**Security Implications**

Malicious users could input text that could change the meaning of the SQL query to expose data, gain administrative access or drop tables.

**Example**

The invocation of the executeQuery method below would be flagged because the argument is constructed from values that might include user input.

```
String query =
"SELECT userid, name FROM user_data WHERE accountnum = '"
+ req.getParameter("ACCT_NUM")
+ "'";
...
statement.executeQuery(query);
```

It should be replaced by something like the following:

```
PreparedStatement statement = connection.prepareStatement
("SELECT userid, name FROM user_data WHERE accountnum = ?");
statement.setString(1, req.getParameter("ACCT_NUM");
ResultSet results = statement.executeQuery();
```

**Avoid Class Loaders in EJB**

**Severity:** Medium

**Summary**

The enterprise bean must not attempt to create a class loader, obtain the current class loader, or set the context class loader.

**Description**

According to the EJB specification, the enterprise bean must not attempt to create a class loader, obtain the current class loader, or set the context class loader. These functions are reserved for the EJB container.

**Security Implications**

Allowing the enterprise bean to use these functions could compromise security and decrease the Container's ability to properly manage the runtime environment.

**Example**

The following code will be marked as a violation because it accesses a class loader in an EJB EntityBean:

```
public class MyBean implements EntityBean {
public void ejbActivate() throws EJBException, RemoteException {
ClassLoader classLoader = MyBean.class.getClassLoader();
}
...
}
```

**Avoid Comparing Classes By String Names**

**Severity:** Medium

**Summary**

String comparisons should not occur with the output from Class.getName()

**Description**

This audit rule looks for places where a class name is compared using the methods String.equals or String.equalsIgnoreCase, or the == or != operators.

Specifically, this audit rule flags the following patterns:

```
[class].getName().equals(*)
*.equals([class].getName())
[class].getName().equalsIgnoreCase(*)
*.equalsIgnoreCase([class].getName())
[class].getName() == *
* == [class].getName()
[class].getName() != *
* != [class].getName()
```

Where [class] is any instance of `java.lang.Class`.

**Security Implications**

By not making comparisons in this way, code is prevented from malicious users creating a class with the same name in order to gain access to blocks of code not intended by the programmer.

**Example**

The following method invocation of equals would be flagged a violation:

```
if ("SomeClassName".equals(class.getName())) ...
```

**Avoid Fields in Action Classes**

**Severity:** Medium

**Summary**

Fields must not be used in action classes.

**Description**

This audit rule looks for fields in action classes. It will allow fields that are marked as "final", but other fields, whether instance or static, are flagged.

**Example**

In the class below, the field "count" would be flagged:

```
public class MyAction extends Action {
private int count;
...
}
```

**Avoid File IO in EJB**

**Severity:** Medium

**Summary**

An enterprise bean must not use the java.io package to attempt to access files and directories in the file system.

**Description**

According to the EJB specification, an enterprise bean must not use the java.io package to attempt to access files and directories in the file system. The file system APIs are not well-suited for business components to access data. Business components should use a resource manager API, such as JDBC, to store data.

**Example**

The following code will be marked as a violation because it accesses a file loader in an EJB context:

```
public class MyBean implements EntityBean {
public void ejbActivate() throws EJBException, RemoteException {
try {
FileInputStream fis = new FileInputStream("test.txt");
...
}
...
}
```

**Avoid Finalizers**

**Severity:** Medium

**Summary**

Avoid finalizers.

**Description**

Finalizers should be avoided because they can lead to obscure bugs and apparent consumption of operating system resources.

**Example**

Any method with the signature "finalize()" will be flagged.

**Avoid Future Keywords**

**Severity:** Medium

**Summary**

Avoid using names that conflict with future keywords.

**Description**

Words that will be keywords in later versions of Java should not be used as an identifier. Otherwise, you will have to rewrite the code in order to migrate.

**Example**

Any variable, method, or type named "assert" or "enum" will be flagged.

**Avoid GUI in EJB**

**Severity:** Medium

**Summary**

An enterprise bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.

**Description**

According to the EJB specification, an enterprise bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard. Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system.

**Example**

The following code will be marked as a violation because it uses AWT GUI in an EJB context:

```
public class MyBean implements EntityBean {
public void ejbActivate() throws EJBException, RemoteException {
Frame f = new Frame();
}
...
}
```

**Avoid Inner Classes**

**Severity:** Medium

**Summary**

Avoid using inner classes.

**Description**

Inner classes are flagged and considered a security hazard by this audit rule.

**Security Implications**

Inner classes are converted into bytecode just like parent classes. Don't depend on an inner class to keep adversaries from private fields. Remember, an inner class has bytecode separate from parent class, but still has access to private fields in the parent class.

**Example**

The following inner class declaration and the anonymous class declaration would both be flagged as violations:

```
public class OuterClass {

public class InnerClass{}

public void foo() {
Runnable runnable = new Runnable() {
public void run(){/*do nothing*/}
}

}
```

**Avoid Instance Initializers**

**Severity:** Low

**Summary**

Avoid using instance initializers.

**Description**

Java allows the use of instance initializers to run blocks of code before an object's constructor. However, this is not a well known feature of the Java language, and it may make maintenance harder. Also, there is potential to violate encapsulation. This rule prohibits the use of these instance initializers.

**Example**

The following would be flagged as a violation:

```
public class Foo {
{
...
}
}
```

**Avoid Instantiation in Loops**

**Severity:** Low

**Summary**

Avoid instantiating classes in loops.

**Description**

Instantiation of a class requires memory allocation. If a class is instantiated within a loop, memory allocation will be performed over and over again. If the objects are kept, this will eat up memory; if they are abandoned, it will cause excessive garbage collection.
However, this rule allows instantiation in return statements and throw statements, since they will not be repeated. It also allows instantiation in catch blocks, since these should not be reached regularly.

**Example**

The following would be flagged as a violation:

```
while (a < b) {
String s = new String();
}
```

**Avoid Instantiation to Get Class**

**Severity:** Medium

**Summary**

Avoid instantiating an object just to call `getClass()`.

**Description**

It is unnecessary to create a new instance of a class just to call its `getClass()` method. The public member `class` can be accessed without instantiation.

**Example**

The following would be flagged as a violation, since `Object.class` would be more efficient:

```
new Object().getClass();
```

**Avoid Loading Native Libraries**

**Severity:** Medium

**Summary**

Try to avoid using native code in EJB context if possible.

**Description**

This audit rule looks for places where native libraries are loaded in EJB context.

**Security Implications**

Native code is vulnurable to a lot of native-level attacks like buffer overflow. Its usage in security-sensitive contexts should be avoided if possible.

**Example**

The following code will be marked as a violation because native libraries loaded in this case:

```
public class ClassOne implements EntityBean {
public void loadNative(){
System.loadLibrary("Library Name");
}
....
}
```

**Avoid Managing Threads**

**Severity:** Medium

**Summary**

Managing threads manually can cause errors and unpredictable behavior of a container.

**Description**

Thread management in a J2EE environment is a responsibility of an application server. This rule looks for the direct usage and management of threads in J2EE `SessionBean`s and `EntityBean`s.

**Security Implications**

Managing threads manually can cause errors and unpredictable behavior of a container.

**Example**

The following code will be marked as a violation because it starts a new Thread in an EJB context:

```
public class MyBean implements EntityBean {
public void ejbActivate() throws EJBException, RemoteException {
new Thread().start();
}
...
}
```

**Avoid Nested Assignments**

**Severity:** Medium

**Summary**

Assignments should not be nested.

**Description**

This audit rule finds assignments nested within other assignments.

**Example**

The assignment to the variable 'i' would be flagged as a violation in the following statement:

```
int a = (i = j * k) / 2;
```

**Avoid Nested Blocks**

**Severity:** Medium

**Summary**

Blocks should not be nested.

**Description**

This audit rule finds blocks nested directly within other blocks.

**Example**

```
{
...
{
...
}
...
}
```

**Avoid null Return Values**

**Severity:** Medium

**Summary**

Return values should not be null.

**Description**

This audit rule finds places where null is returned rather than array types or simple types.

**Example**

The return statement in the following method would be flagged as a violation:

```
public int[] getRowSums()
{
```

```
if (table == null) {
return null;
}
...
}
```

**Avoid Octal Literals**

**Severity:** Low

**Summary**

Avoid the use of octal literals.

**Description**

This audit rule finds uses of octal literals (numeric literals that begin with a zero). Numeric literals should be expressed in either decimal or hexidecimal formats in order to avoid confusion.

**Example**

The following numeric literal would be flagged as a violation:

```
0010
```

**Avoid Package Scope**

**Severity:** Medium

**Summary**

Only use public, protected or private scopes.

**Description**

This audit flags all inner classes, constructors, methods, and fields that have a package scope.

Note: non-inner classes and interfaces cannot syntactically be declared protected or private, thus since a non-inner class/ interface isn't more secure if it has package scope instead of a public scope, this audit does not flag interfaces or non-inner classes that have a package scope.

Also note: the resolutions (fixes) for flagged instances of this audit include the insertion of the "public" modifier as well as "private" and "protected" modifiers. However, changing a modifier from package scope to a public scope does not make the code more secure, and is included only because it is expected that the public modifier is used.

**Security Implications**

Classes, methods and fields with package scope (default scope) can be accessed from all code within the same package, including code written by adversaries.

**Example**

The following will all be flagged since they all have a package scope: the constructor "Example", the class "InnerClass", the method "some_method", and the integer "x":

```
class Example {
Example(){super();}
class InnerClass{}
static void some_method(){/* do nothing */}
int x;
}
```

**Avoid Passing this Reference as Argument**

**Severity:** Medium

**Summary**

You should not pass a reference to `this` as an argument or as a return value in EJB context.

**Description**

This audit rule looks for a places in the code where a reference to `this` is returned or passed as an argument.

**Security Implications**

Direct passing of a reference to an EJB bean is an incorrect usage of EJB which could lead to security issues. Use `getEJBObject()' available in SessionContext or EntityContext instead. See "Programming restrictions on EJB" (http://www.javaworld.com/javaworld/jw-08-2000/jw-0825-ejbrestrict.html) for details.

**Example**

The following code will be marked as a violation because it attempts to return a this reference:

```
public class ClassOne implements EntityBean {
public EntityBean getBean(){
return this;
}
....
}
```

**Avoid Primitive Method Parameters**

**Severity:** Medium

**Summary**

Don't use primitive types for method arguments.

**Description**

Prohibit the use of primitive types in method parameters. All parameters in methods should be Java objects. Exceptions are given to methods that override a superclass method that uses a primitive type as a parameter.

**Example**

The following method declaration, assuming that it does not override an inherited method, would be flagged as a violation because it has a parameter that is a primitive type:

```
public void setCount(int count)
```

**Avoid Similar Names**

**Severity:** Low

**Summary**

Avoid names that are too similar.

**Description**

This rule detects pairs of names that are too similar and therefore can be confusing to read. Names that are spelled the same but are distinct because some letters have different case are too similar in most contexts. Using the singular and plural form of a word as separate names can also be misleading in some contexts. This rule also detects possible spelling errors where two identifiers are almost the same but have just a slight difference in spelling.

**Example**

```
private String customername;

public String customerName()
{
return customername;
}
```

**Avoid Sockets in EJB**

**Severity:** Medium

**Summary**

An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.

**Description**

According to the EJB specification, an enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast. The EJB architecture allows an enterprise bean instance to be a network socket client, but it does not allow it to be a network server. Allowing the instance to become a network server would conflict with the basic function of the enterprise bean - to serve the EJB clients.

**Example**

The following code will be marked as a violation because it attempts to accept a socket connection in an EJB context:

```
public class MyBean implements EntityBean {
public void ejbActivate() throws EJBException, RemoteException {
try {
ServerSocket ss = new ServerSocket(8080);
ss.accept();
...
}
```

```
...
}
```

**Avoid StringBuffer Instantiation With Character Literal**

**Severity:** Medium

**Summary**

Character literals should not be used as the argument to the constructor of either a StringBuffer or StringBuilder.

**Description**

This audit rule finds places in the code where a character literal is used to initialize a newly created StringBuffer or StringBuilder. This has the (presumably) unintended result of converting the character literal into an int to set the initial size of the StringBuffer or StringBuilder.

**Example**

The following expression would be flagged as a violation:

```
new StringBuffer('c');
```

**Avoid Subtyping Cloneable**

**Severity:** Medium

**Summary**

It is best to avoid creating subtypes of Cloneable.

**Description**

The Cloneable interface serves no purpose other than to modify the behavior of Object's clone method. There's no reason for interfaces to extend it. Classes rarely benefit from implementing it; there are better ways to copy objects than by cloning.

**Example**

The following class declaration would be flagged as a violation:

```
public class Customer implements Cloneable
```

**Avoid Synchronization in EJB**

**Severity:** Medium

**Summary**

An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances.

**Description**

According to the EJB specification, an enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances. Synchronization would not work if the EJB container distributed the enterprise bean's instances across multiple JVMs.

**Example**

The following code will be marked as a violation because it uses `synchronize` mechanism in an EJB context:

```
public class MyBean implements EntityBean {
public void ejbActivate() throws EJBException, RemoteException {
synchronize (this) {
...
}
}
...
}
```

**Avoid the no-argument String constructor**

**Severity:** Medium

**Summary**

Avoid using the no-argument String constructor.

**Description**

The no-argument String constructor `new String()`, creates a String that will not be functionally

different from the empty String (""). Since Java guarantees that identical String constants will all be the same instance, you can improve performance by using the empty String instead.

**Example**

The following would be flagged as a violation:

```
String foo = new String();
```

This could be replaced with the following, which would conserve memory:

```
String foo = "";
```

**Avoid Unsafe Array Declaration**

**Severity:** Medium

**Summary**

Refrain from declaring an array public, static and final.

**Description**

This audit rule flags an array declared public, static and final. Arrays are mutable objects, the final constraint requires that the array object itself be assigned only once, but makes no guarantees about the values of the array elements. Since the array is declared public, a malicious program can change the values stored in the array.

**Security Implications**

Such arrays are many times assumed to be secure and thus used by the application as a secure field. Malicious users can seek out such fields to exploit.

**Example**

The following Java Applet code would be flagged as a violation because it declares an array public, static and final:

```
public final class UrlTool extends Applet {
public static final URL[] urls;
...
}
```

**Avoid Using "Field Access" Strategy**

**Severity:** Medium

**Summary**

Access to properties via accessor methods is considered best practice by the Hibernate community.

**Description**

The access attribute allows you to specify how Hibernate should access property values of the POJO. The default strategy, "property", uses the property accessors (get/set method pair). The "field" strategy uses reflection to access the instance variable directly. Access to properties via accessor methods is considered best practice by the Hibernate community. It provides an extra level of abstraction between the Java domain model and the data model, beyond what is already provided by Hibernate. Properties are more flexible; for example, property definitions may be overridden by persistent subclasses.

This rule will only create a violation if both accessor methods are defined for the field. You can use the Declare Accessors for Persistent Fields rule to ensure that accessor methods exist.

**Example**

The value of the "access" attribute would be flagged:

```
<property name="name" column="NAME" access="field">
```

**Avoid Using "instanceof"**

**Severity:** Medium

**Summary**

Use polymorphism instead of instanceof (except when used with interfaces).

**Description**

This audit rule looks for uses of "instanceof". In general, one should use polymorphism instead of instanceof. Optionally, require that "instanceof" only be used on interfaces.

**Example**

The following uses of the instanceof operator would be flagged as violations:

```
if (employee instanceof AccountingEmployee) {
return "Accounting";
} else if (employee instanceof DevelopmentEmployee) {
return "Development";
...
```

**Avoid Using Autowiring**

**Severity:** Medium

**Summary**

Avoid using autowiring of dependencies through introspection of the bean classes.

**Description**

This audit rule looks for places where dependencies are being autowired. Spring can autowire dependencies through introspection of the bean classes so that you do not have to explicitly specify the bean properties or constructor arguments. Bean properties can be autowired either by property names or matching types. Constructor arguments can be autowired by matching types. Autowiring can potentially save some typing and reduce clutter.

However, you should not use it in real-world projects because it sacrifices the explicitness and maintainability of the configurations. Autowiring seems like a good idea to make the XML configuration file smaller, but this will actually increase the complexity down the road, especially when you are working on a large project where many beans are defined. Spring allows you to mix autowiring and explicit wiring, but the inconsistency will make the XML configurations even more confusing.

**Example**

The following bean declaration would be flagged:

```
<bean id="orderService"
class="com.test.spring.OrderService"
autowire="byName"/>
```

**Avoid Using Enterprise Schemas Version**

**Severity:** Medium

**Summary**

Enterprise schemas should not contain version numbers within the file names.

**Description**

This audit rule looks for enterprise schemas that contain a version number in their name.

**Example**

The following enterprise schemas would be flagged as a violation because it contains a version number:

```
Enterprise_Quote_v1.92.xsd
```

It should be renamed to:

```
Enterprise_Quote.xsd
```

**Avoid Utility Methods**

**Severity:** Medium

**Summary**

Utility methods should be avoided except under certain circumstances.

**Description**

A utility method is defined to be any method that is not required to be defined in any particular location. For example, a method to compute the sum of the elements in an array of integers doesn't refer to anything other than the parameter, so the class in which it is declared is irrelevent. Utility methods are usually a strong indication that behavior has not been implemented where it should be, and are therefore an indication that the code needs to be refactored.

**Badly Located Array Declarators**

**Severity:** Low

**Summary**

Array declarators should be placed next to the type descriptor of their component type.

**Description**

This audit rule checks to ensure that the array declarators (the "[]" in the declaration of a variable with an array type) occur after the type name rather than after the variable name.

**Example**

The following declaration would be flagged as a violation because of the placement of the array brackets:

```
int x[];
```

It should be replaced by a declaration of the form:

```
int[] x;
```

**Base64-Encoded Password**

**Severity:** Medium

**Summary**

A password protected with a trivial encoding is as easy for an attacker to read as a plain text one.

**Description**

This audit rule violates all the usages of Base64 encoding to decode passwords.

**Security Implications**

Base64 is a common, familiar and easily decrypted encoding. Using it for password encoding provides no real protection. You should use more strong encodings instead.

**Example**

The following code will be marked as a violation because it uses Base64 to decode a password read from a file:

```
String password = props.getProperty("password");
byte[] decodedPassword = Base64.decodeBase64(password.getBytes());
```

**Bean Members Should Be Serializable**

**Severity:** Medium

**Summary**

Members of beans should be marked as transient, static or should have accessors.

**Description**

This audit rule looks for bean members that are neither `static`, `transient` nor have acessors.

**Security Implications**

If a class is a bean, or is referenced by a bean directly or indirectly it needs to be serializable. Member variables need to be marked as transient, static, or have accessor methods in the class.

**Example**

The following declaraion of `badFoo` field will be marked as a violation because it is not static or transient and it does not have accessors:

```
....
private transient int someFoo;
private static int otherFoo;
private int moreFoo;
private int badFoo;
....
private void setMoreFoo(int moreFoo) {
this.moreFoo = moreFoo;
}
private int getMoreFoo() {
return this.moreFoo;
}
```

**Beware of URL equals() and hashCode()**

**Severity:** Medium

**Summary**

Be careful when and how you use the `equals()` and `hashCode()` methods of the `URL` class.

**Description**

Both the `equals()` and `hashCode()` methods of the `URL` class resolve the domain name using the Internet. This operation can cause unexpected performance problems. Also, the `hashCode()` method takes the resolved IP address into account when generating the hash code. This can cause serious problems since many web sites use dynamic DNS. It is possible to store a `URL` in a hashed collection, and later be unable to retrieve it if the `URL` resolves to a different IP address.

Because of these implementation problems, it is a good idea to convert `URL`s to `URI`s before storing them in collections, or using their `equals()` or `hashCode()` methods. This can be done easily using `URL`'s `toURI()` method, and reversed using `URI`'s `toURL()` method.

This rule finds places where `equals()` or `hashCode()` are explicitly invoked on `URL` objects and places where URL objects are used in hashed Collections classes.

**Example**

The following would be flagged as a violation:

```
URL aUrl = new URL("http://address.com");
Set aSet = new HashSet();
aSet.add(aUrl);
```

**Blank Line Usage**

**Severity:** Low

**Summary**

Blank lines should be used consistently to improve readability.

**Description**

This audit rule checks for places where the number of blank lines used between program elements is not consistent.

**Example**

If the rule has been configured to expect two blank lines between members in a type, then the following field declarations will be flagged because there is only one blank line between them:

```
private int x;

private int y;
```

**Blank Password**

**Severity:** Medium

**Summary**

Blank passwords are a security threat that can be used by an attacker to access secure information.

**Description**

This audit rule violates blank password properties in a property files.

**Security Implications**

A blank password is usually the first one to be tried when trying to guess or brute-force a password protection. It should never be used.

**Example**

The following entry in the property file will be marked as a violation because it declares a blank password:

```
invalidPassword=
```

**Block Depth**

**Severity:** Low

**Summary**

Methods should be kept fairly flat.

**Description**

This audit rule finds methods and constructors that have too many levels of nested blocks. A method with too many levels of nested blocks can be difficult to understand. The definition of "too many" can be set.

**Example**

If the limit is set to two, then the inner if statement in the following would be flagged as a violation:

```
if (firstChoice == null) {
if (secondChoice == null) {
if (thirdChoice == null) {
return null;
}
}
}
```

**Boolean Method Naming Convention**

**Severity:** Medium

**Summary**

Boolean method names should start with 'is', 'can', 'has' or 'have'. Non-boolean methods should not start with 'is'.

**Description**

This audit rule checks the names of all boolean methods to make sure that they are prefixed with 'is', 'can', 'has' or 'have'. It also checks the names of all non-boolean methods to make sure that they are not prefixed with 'is'.

**Example**

The following method would be flagged as a violation because it returns a boolean value but does not start with an approved prefix:

```
public boolean willExplodeIfShaken()
```

The following method would be flagged as a violation because it starts with 'is' but does not return a boolean:

```
public int isRoundedBy()
```

**Brace Position**

**Severity:** Low

**Summary**

Opening and closing braces should be positioned properly.

**Description**

This audit rule finds opening and closing braces that are incorrectly positioned. By default, opening braces should not appear on a new line and closing braces should appear on a new line. Opening braces for methods and types should appear on a new line.

**Example**

With the default settings, the opening brace for the following method and the opening brace for the if statement within it would both be flagged as violations:

```
public int getLength(Object[] array) {
if (array == null)
{
return 0;
}
return array.length;
}
```

**Break with Label**

**Severity:** Medium

**Summary**

Break statements should not reference a labeled statement.

**Description**

This audit rule finds break statements that reference a labeled statement. The use of a label with a break statement makes the code much harder to read and maintain, and should therefore be avoided. Consider moving the code that contains the break into a separate method and using a return statement instead.

**Example**

The following break statement would be flagged as a violation:

```
outer: for (int i = 0; i < array.length; i++) {
for (int j = 0; j < array[i].length; j++) {
if (array[i][j] == 0) {
break outer;
}
...
}
}
```

**Bundle Activation Policy Compatibility (3.3)**

**Severity:** Medium

**Summary**

The bundle activation policy must be duplicated in order to work in all versions of Eclipse.

**Description**

This audit rule checks for bundle manifest files that specify a bundle activation policy using either the `Eclipse-LazyStart` or `Bundle-ActivationPolicy` header, but not both. In Eclipse 3.3 a new manifest file header was introduced that is compatible with OSGI. In order for a bundle (plug-in) to be correct in all versions of Eclipse (that support the use of manifest files) both headers should be included in the manifest file.

**Example**

A manifest file specifying a `Bundle-ActivationPolicy` without a `Eclipse-LazyStart` would be flagged:

```
Bundle-ActivationPolicy: lazy
```

**Business Logic in ActionForm**

**Severity:** Medium

**Summary**

ActionForms should not contain any business logic.

**Description**

This audit rule looks for subclasses of ActionForm that contain business logic. ActionForms should only be used to keep form data passed between the view and the controller. An ActionForm has business logic if it contains any methods other than getters, setters and a few other specific methods (validate, reset, toSession, and fromSession). Business logic should be moved into the model.

**Example**

In the following class, the method generatePassword would be flagged as business logic that needs to be moved.

```
public class UserFormBean extends ActionForm {
private String email = "Missing address";
private String password = "Missing password";

public String getEmail() {
return(email);
}

public void setEmail(String email) {
this.email = email;
}

public String getPassword() {
return(password);
}

public void setPassword(String password) {
this.password = password;
}

public void generatePassword() {
Random r = new Random(System.currentTimeMillis());
int len = 6 + r.nextInt() % 12;
byte[] pass = new byte[len];
for(int i = 0; i < len; i++) {
...
}
...
}
}
```

**Call Lock Without Unlock**

**Severity:** Medium

**Summary**

Invoke unlock() method in catch clause or finally block.

**Description**

This audit rule looks for places where the `lock()` method is invoked without guaranteeing that the `unlock()` method will be invoked if an exception is thrown.

**Security Implications**

Such a mistake can cause a deadlock in the case of an exception. Deadlocks are a common way of depleting a web application's pool of threads and thus permitting a denial-of-service attack.

**Example**

The following invocation of the `lock()` method will be flagged as a violation because the `unlock()` method is not invoked if an exception is thrown:

```
public void myMethod()
{
.......
Lock l = new Lock();
l.lock();
try{
iter.next();
}catch(Exception e) {
}
l.unlock();
}
```

**Caught Exceptions**

**Severity:** Medium

**Summary**

Some exceptions should not be caught.

**Description**

This audit rule finds catch clauses that catch an exception class that is disallowed. The list initially includes exception classes that are either too general (such as Throwable or Exception), or that are unchecked (such as Error and RuntimeException).

**Example**

The following catch clause would be flagged because it catches instances of the class java.lang.Throwable, which is too general:

```
try {
...
} catch (Throwable exception) {
...
}
```

**Character Comparison**

**Severity:** Medium

**Summary**

Character values should not be compared using any of the non-equality operators.

**Description**

This audit rule finds places where two Character values are compared using any of the non-equality operators. Such comparisons are usually an attempt to classify characters and such tests are usually not valid across all locales.

**Example**

The following code, designed to test for a lower-case letter, would be flagged as a violation:

```
if (ch >= 'a' && ch <= 'z') {
...
}
```

**Check Enabled Before Logging**

**Severity:** Medium

**Summary**

Check that logging has been enabled before invoking the logging methods.

**Description**

This audit rule finds invocations of the Log4J logging methods that are not guarded by a check to ensure that the appropriate level of logging has been enabled. Invocations of the logging methods should be guarded to avoid computation of logging output when it isn't going to be reported.

**Example**

The following invocation of the debug method:

```
logger.debug("I never thought we'd get here!");
```

should be replaced by something like the following:

```
if (logger.isDebugEnabled()) {
logger.debug("I never thought we'd get here!");
}
```

**Check Enabled Before Logging**

**Severity:** Medium

**Summary**

Check that logging has been enabled before invoking the logging methods.

**Description**

This audit rule finds invocations of the JCL logging methods that are not guarded by a check to ensure that the appropriate level of logging has been enabled. Invocations of the logging methods should be guarded to avoid computation of logging output when it isn't going to be reported.

**Example**

The following invocation of the debug method:

```
logger.debug("I never thought we'd get here!");
```

should be replaced by something like the following:

```
if (logger.isDebugEnabled()) {
logger.debug("I never thought we'd get here!");
}
```

**Check Type In Equals**

**Severity:** Medium

**Summary**

Implementations of equals() should check the type of the parameter.

**Description**

This audit rule finds implementations of the method equals() that do not check the type of the parameter. The rule can be configured for how the type of the parameter should be checked.

**Example**

The following declaration of the equals() method would be flagged because the type of the argument is not checked:

```
public boolean equals(Object other)
{
return getName().equals(((Employee) other).getName());
}
```

**Class Extends java.security.Policy**

**Severity:** Medium

**Summary**

Classes should not extend `java.security.Policy`.

**Description**

This audit rule looks for classes that subclass the class `java.security.Policy`.

**Security Implications**

Allowing an implementation of `java.security.Policy` could lead to a security (and/or permission) breech.

**Example**

The following class would be flagged as a violation because it extends `java.security.Policy`:

```
import java.security.Policy;

class MyClass extends Policy
{
...
}
```

**Class getName() Usage**

**Severity:** Medium

**Summary**

Don't use the Class getName() method.

**Description**

Don't use the Class getName() method to compare classes.

**Example**

The following invocation of getName() would be flagged as a violation:

```
if (object.getClass().getName().equals("java.util.ArrayList")) {
```

**Class Naming Convention**

**Severity:** Medium

**Summary**

Class names should conform to the defined standard.

**Description**

This audit rule checks the names of all classes to ensure that they conform to the standard.

**Example**

If class names are specified as needing to start with an upper-case letter, the following class definition would be flagged as being a violation:

```
public class remoteDatabase
{
...
}
```

**Class Should Be Final**

**Severity:** Medium

**Summary**

Classes that do not have subclasses should be `final`.

**Description**

This audit rule looks for classes that do not have any subclasses but are not marked as `final`. Note that this rule is only useful if all of the classes in the product are loaded into the development environment.

**Security Implications**

The ability to extend classes can be abused by a malicious user. They can be used to obtain access to information contained in the parent class.

**Example**

The following class would be flagged as a violation because it does not have any subclasses, and it isn't marked as `final`:

```
class MyClass
{
...
}
```

**Class Should Define Validate Method**

**Severity:** Medium

**Summary**

If a class extends `org.apache.struts.action.ActionForm`, it must define a `validate` method.

**Description**

This audit rule looks for subclasses of `ActionForm` that do not implement a `validate` method. The validate method is the standard mechanism for a Struts framework to validate fields.

**Security Implications**

Fields that are not validated are security problem number one for web applications. Failing to validate fields can lead to many security vulnerabilities such as SQL injections and Cross-site scripting.

**Example**

The following class would be flagged as a violation because the class in this example doesn't define `validate` method:

```
class MyClass extends ActionForm
{
...
}
```

**Class Should Have Final Static Fields**

**Severity:** Medium

**Summary**

If a class extends `org.apache.struts.action.ActionForm`, it must not contains non-final static fields.

**Description**

This audit rule looks for subclasses of `ActionForm` that define non-final static fields.

**Security Implications**

Non-final static fields can be changed by different instances of the form at the same time (on

different threads), or between the time of change and the time of use. This fact can provide an opportunity for malicious code to make unsafe changes even if the field is private.

**Example**

The following field would be flagged as a violation because it is static but is not final:

```
class MyClass extends ActionForm
{
public static String testStr = "Value";
}
```

**Class Should Have Private Fields**

**Severity:** Medium

**Summary**

If a class extends `org.apache.struts.action.ActionForm`, it must contain only private fields.

**Description**

This audit rule looks for subclasses of `ActionForm` that define non-private fields that are accessed by getters and setters. The setter method should be used only by the framework; setting an action form field from other actions is bad practice and should be avoided.

**Security Implications**

Non-private fields can be accessed in a way that avoids validation and creates an invalid state for the Form object. This can be used by malicious code to create unexpected and unintended behavior.

**Example**

The following field would be flagged as a violation because it is not private:

```
class MyClass extends ActionForm
{
public static testStr = "Value";
}
```

**Class Should Validate All Fields**

**Severity:** Medium

**Summary**

If a class extends `org.apache.struts.action.ActionForm`, it should validate all of the fields it declares.

**Description**

This audit rule looks for subclasses of `ActionForm` in which the validate method does not validate all of the fields defined by the class.

**Security Implications**

Unvalidated fields is the number one security issue for web-applications. Unvalidated fields can lead to many security vulnerabilities such as SQL Injections and Cross Site scripting.

**Example**

The following class would be flagged as a violation because the field `count` is not validated by `validate` method.

```
class MyClass extends ActionForm
{
private testStr = "Value";
private int count;

public ActionErrors validate(ActionMapping map,
ServletRequest req) {
ActionErrors err ActionErroes();
if (testStr == null) {
err.add(ActionErrors.GLOBAL_MESSAGE,
new ActionMessage("testStr is null"));
}
return super(map, req);
}
}
```

**Classes Should be Their Own Proxy**

**Severity:** Medium

**Summary**

Every persistent class should be its own proxy.

**Description**

This audit rule looks for declarations of persistent classes in which a proxy is specified that is different from the class itself. The Hibernate framework recommends that every persistent class should be its own proxy.

**Example**

The following tag would be flagged because the proxy class is not the same as the class being described

```
<class name="eg.Order" proxy="eg.IOrder">
```

It should be replaced by the following

```
<class name="eg.Order" proxy="eg.Order">
```

**Client Request Not Encrypted**

**Severity:** Medium

**Summary**

Websphere web service client should only accept encrypted messages.

**Description**

This audit rule looks for declarations of Websphere web service client extensions and violates the declarations which do not declare message encryption.

**Security Implications**

Message-level encryption is especially important for messages that are passed via such easily readable protocols as HTTP or SMTP, as well as the others. Encrypted message maintains the required level of security not depending on the protocol.

**Example**

The following code specifies the encryption constraints that messages consumed by a service client must meet. If no encryption constraints, declared via `<confidentiality>` property, are defined, the whole `<securityRequestSenderServiceConfig>` declaration is marked as violation:

```
<securityRequestSenderServiceConfig>
<confidentiality>
<confidentialParts part="bodycontent"/>
</confidentiality>
...
</securityRequestSenderServiceConfig>
```

**Client Request Not Signed**

**Severity:** Medium

**Summary**

Websphere web service client should only accept signed messages.

**Description**

This audit rule looks for declarations of Websphere web service client extensions and violates the declarations which do not declare signing the message.

**Security Implications**

Signing the message is important because this is the only way to provide the complete guarantee for the receiver of a message that the message was endeed sent by a specific sender. Otherwise a message could be intercepted and its contents could be changed.

**Example**

The following code verifies that that messages consumed by a service client are signed. If they are not verified to be signed, as declared via `<integrity>` property, the whole `<securityRequestSenderServiceConfig>` declaration is marked as violation:

```
<securityRequestSenderServiceConfig>
<integrity>
<references part="body"/>
<references part="timestamp"/>
</integrity>
```

```
...
</securityRequestSenderServiceConfig>
```

**Client Request Timestamp Not Signed**

**Severity:** Medium

**Summary**

Websphere web service timestamps should be signed.

**Description**

This audit rule looks for declarations of Websphere web service requester extensions and violates the declarations which do not declare signing of timestamps.

**Security Implications**

Unsigned timestamp can be intercepted and replaced by an attacker, thus easing a task of forging the message.

**Example**

The following code declares a `<addCreatedTimeStamp>` without mentioning it in a `<integrity>` list of signed components; it would thus be marked as violation:

```
<securityRequestSenderServiceConfig>
<integrity>
<references part="body"/>
</integrity>
<addCreatedTimeStamp flag="true"/>
...
</securityRequestSenderServiceConfig>
```

**Client Response Not Encrypted**

**Severity:** Medium

**Summary**

Websphere web service client should only produce encrypted messages.

**Description**

This audit rule looks for declarations of Websphere web service client extensions and violates the declarations which do not declare message encryption.

**Security Implications**

Message-level encryption is especially important for messages that are passed via such easily readable protocols as HTTP or SMTP, as well as the others. Encrypted message maintains the required level of security not depending on the protocol.

**Example**

The following code specifies the encryption constraints that messages produced by a service client must meet. If no encryption constraints, declared via `<requiredConfidentiality>` property, are defined, the whole `<securityResponseReceiverServiceConfig>` declaration is marked as violation:

```
<securityResponseReceiverServiceConfig>
<requiredConfidentiality>
<confidentialParts part="bodycontent"/>
<confidentialParts part="usernametoken"/>
</requiredConfidentiality>
...
</securityResponseReceiverServiceConfig>
```

**Client Response Not Signed**

**Severity:** Medium

**Summary**

Websphere web service client should only produce signed messages.

**Description**

This audit rule looks for declarations of Websphere web service client extensions and violates the declarations which do not declare signing the message.

**Security Implications**

Signing the message is important because this is the only way to provide the complete guarantee

for the receiver of a message that the message was indeed sent by a specific sender. Otherwise a message could be intercepted and its contents could be changed.

**Example**

The following code verifies that that messages consumed by a service client are signed. If they are not verified to be signed, as declared via `<requiredIntegrity>` property, the whole `<securityResponseReceiverServiceConfig>` declaration is marked as violation:

```
<securityResponseReceiverServiceConfig>
<requiredIntegrity>
<references part="body"/>
<references part="timestamp"/>
</requiredIntegrity>
...
</securityResponseReceiverServiceConfig>
```

**Client Response Timestamp Not Signed**

**Severity:** Medium

**Summary**

Websphere web service timestamps should be signed.

**Description**

This audit rule looks for declarations of Websphere web service requester extensions and violates the declarations which do not declare signing of timestamps.

**Security Implications**

Unsigned timestamp can be intercepted and replaced by an attacker, thus easing a task of forging the message.

**Example**

The following code declares a `<addReceivedTimeStamp>` without mentioning it in a `<requiredIntegrity>` list of signed components; it would thus be marked as violation:

```
<securityResponseReceiverServiceConfig>
<requiredIntegrity>
<references part="body"/>
```

```
</requiredIntegrity>
<addReceivedTimeStamp flag="true"/>
...
</securityResponseReceiverServiceConfig>
```

**Client Timestamp Does Not Expire**

**Severity:** Medium

**Summary**

Websphere web service client timestamps should have an expiration date.

**Description**

This audit rule looks for declarations of Websphere web service client extensions and violates the declarations which do not declare expiration date for received timestamps.

**Security Implications**

Timestamp without an expiration date is valid indefinetely, thus allowing the attacker to delay and modify messages, perform replay attacks, etc.

**Example**

The following code declares a `<addCreatedTimeStamp>` without an expiration date set through a `expires` property; it would thus be marked as violation:

```
<securityRequestSenderServiceConfig>
<integrity>
<references part="body"/>
<references part="timestamp"/>
</integrity>
<addCreatedTimeStamp flag="true"/>
...
</securityRequestSenderServiceConfig>
```

**Client Uses Username Token**

**Severity:** Medium

**Summary**

UsernameToken authentication is insecure and should not be used.

**Description**

This audit rule looks for declarations of Websphere web service client extensions and violates the declarations which use UsernameToken for authentication.

**Security Implications**

A UsernameToken authentication could use an unencrypted password that could be intercepted by an attacker.

**Example**

The following code declares a `<securityToken>` with a UsernameToken as an authentication token and would thus be marked as violation:

```
<securityRequestGeneratorServiceConfig>
<securityToken name="basicauth" uri="" localName="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-username-token-profile-
1.0#UsernameToken"/>
...
</securityRequestGeneratorServiceConfig>
```

**Clone Method Usage**

**Severity:** Medium

**Summary**

Every implementation of clone() should invoke super.clone() and every clone() method should be final.

**Description**

This audit rule looks for implementations of the method clone() that do not invoke the inherited clone() method. It also looks for clone() methods that are not declared final.

**Example**

The following declaration of the clone method would be flagged as being a violation, both because of the lack of an invocation of the inherited clone method and because it is not declared 'final'.

```
public Object clone()
{
return new Employee(getName());
}
```

**Clone Without Cloneable**

**Severity:** Medium

**Summary**

Invoking `Object`'s `clone()` method on an instance that does not implement the `Cloneable` interface results in the exception `CloneNotSupportedException` being thrown.

**Description**

The programmer probably intended the class to be cloneable when implementing the `clone()` method. Invoking `Object`'s `clone()` method on an instance that does not implement the `Cloneable` interface results in the exception `CloneNotSupportedException` being thrown.

**Security Implications**

This means the code will not work as intended, resulting in errors and possibly unpredictable behavior thus compromising security.

**Example**

The following code would be flagged as a violation because it does not implement `Cloneable` while implementing `clone` method:

```
public class SomeBean {
public Object clone() throws CloneNotSupportedException {
...
}
}
```

**Cloneable Without Clone**

**Severity:** Medium

**Summary**

Classes that implement the Cloneable interface should define a clone() method.

**Description**

This audit rule looks classes that implement the Cloneable interface and do not define clone() method.

**Security Implications**

It is most essential for library vendors or for mobile code. The `clone` method is mechanism that allows the creation of objects, and it might be undesirable for users to be able to inherit this class and tamper with the behavior of this method.

**Example**

The following class would be flagged as a violation because it does not define a clone() method

```
class MyClass implements Cloneable
{
...
}
```

**Close Connection Where Created**

**Severity:** Medium

**Summary**

Database connections should be closed in the same method in which they are created.

**Description**

This rule finds methods in which a database connection is created but not closed. When the close is in a different method it is harder to ensure that it will always be invoked.

**Example**

The following invocation would be flagged as a violation because there is no corresponding close method:

```
public void readFromDatabase(String url)
{
Connection connection = DriverManager.getConnection(url);
readFromDatabase(connection);
}
```

**Close Elements in Renderer**

**Severity:** Medium

**Summary**

Elements that are opened in a renderer should be closed in the same method.

**Description**

One of the ways a custom renderer (which is derived from javax.faces.render.Renderer) produces markup is by calling startElement/writeAttribute/endElement methods. This audit rule looks for invocations of startElement that are not followed by a proper endElement invocation within the same method.

**Example**

```
public class CompARenderer extends Renderer {
public void encodeEnd(FacesContext context, UIComponent component) throws
IOException {
UIOutput outComp = (UIOutput)component;
ResponseWriter writer = context.getResponseWriter();
writer.startElement("div", component);
writer.writeAttribute("id", clientId, "id");
writer.endElement("p");
}
}
```

**Close In Finally**

**Severity:** Medium

**Summary**

The method close() should be invoked inside a finally block.

**Description**

This rule finds places where the method close() is invoked outside of a finally block.

**Example**

The following invocation would be flagged as a violation because it occurs outside of a finally block:

```
public void readFile(FileReader reader)
{
...
reader.close();
}
```

**Close Order**

**Severity:** Medium

**Summary**

Close ResultSets before Statements and Statements before Connections.

**Description**

This rule find places where a Statement is closed after a Connection or where a ResultSet is closed after either of the other two in a single method.

**Example**

The following closing of the result set would be flagged as a violation:

```
public void close(ResultSet resultSet)
{
resultSet.getStatement().close();
resultSet.close();
}
```

**Close Result Set Where Created**

**Severity:** Medium

**Summary**

Database result sets should be closed in the same method in which they are created.

**Description**

This rule finds methods in which a database result set is created but not closed. When the close is in a different method it is harder to ensure that it will always be invoked.

**Example**

The following invocation would be flagged as a violation because there is no corresponding close method:

```
public void readFromDatabase(Statement statement)
{
ResultSet resultSet = statement.getResultSet();
readFromDatabase(resultSet);
}
```

**Close Sessions Where Opened**

**Severity:** Medium

**Summary**

Close sessions in the method where they are opened.

**Description**

This audit rule looks for methods in which a session is opened but not closed. The method Session.close() should be invoked in the same method as the method SessionFactory.openSession().

**Example**

The following method would be flagged because the opened session isn't closed.

```
private void createMessage(String text, Date date) {
...
Session session = sessionFactory.openSession();
session.beginTransaction();

Message message = new Message();
```

```
message.setText(text);
message.setDate(date);

session.save(message);
session.getTransaction().commit();
}
```

To close the session, the method session.close() should be invoked in a finally block, as shown below

```
private void createMessage(String text, Date date) {
...
Session session = sessionFactory.openSession();
try {
session.beginTransaction();
Message message = new Message();
message.setText(text);
message.setDate(date);

session.save(message);
} finally {
session.close();
}
}
```

**Close Statement Where Created**

**Severity:** Medium

**Summary**

Database statements should be closed in the same method in which they are created.

**Description**

This rule finds methods in which a database statement is created but not closed. When the close is in a different method it is harder to ensure that it will always be invoked.

**Example**

The following invocation would be flagged as a violation because there is no corresponding close method:

```
public void readFromDatabase(Connection connection)
{
Statement statement = connection.createStatement();
readFromDatabase(statement);
}
```

**Close Where Created**

**Severity:** Medium

**Summary**

Streams, readers, writers and sockets should be closed in the method where they are created.

**Description**

Instances of subclasses of java.io.InputStream, java.io.OutputStream, java.io.Reader, java.io.Writer, and java.net.Socket should be closed in the same method in which they are created in order to avoid errors caused when they are not closed at all.

**Example**

The creation of a reader in the following method would be flagged as a violation because the reader is not closed:

```
public void readFile(String filePath)
{
FileReader reader;

reader = new FileReader(new File(filePath));
readFile(reader);
}
```

**Code in Comments**

**Severity:** Medium

**Summary**

Code that has been commented-out should be removed.

**Description**

This rule searches for comments that appear to contain code and flags them. Such code should either be removed or made conditional (for example, by testing a logging level).

**Example**

The following comment would be flagged:

```
//System.out.println("Entering method");
```

**Code Injection**

**Severity:** High

**Summary**

Scripts being executed might be receiving data from the user or other unsafe sources.

**Description**

Code Injection occurs when the user is able to enter data directly into scripts to be executed.

The difference between Code Injection and Command Injection is that Code Injection attempts try to introduce scripts that are then included as part of the application runtime, while Command Injection attacks execute the user input source from outside of the application runtime.

To detect violations, this audit rule searches the code for script executions such as `javax.script.ScriptEngine.eval(..)` and traces where the script data could have come from. In cases where the source of the query is user input, such as data from a servlet request, `javax.servlet.ServletRequest.getParameter(java.lang.String)`, or from a SWT Text widget, `org.eclipse.swt.widgets.Text.getText()`, a violation is created.

These two sets of methods, the locations where tainted user data can come from and the methods used to execute the scripts, are editable by the user.If methods are missing that are in a common package (such as `java.lang.*`), please let CodePro support know.

**Security Implications**

Successful Code Injection attacks can potentially have malicious scripts executed with permissions not guarded against by the program.

**Example**

The invocation of the method `eval(..)` would be flagged as a violation since it uses information passed from a SWT Text widget:

```
Text text = ...;
String script = text.getText();
```

```
ScriptEngine scriptEngine = ...;
...
scriptEngine.eval(script);
```

**Command Execution**

**Severity:** Medium

**Summary**

External commands should not be executed because not all platforms have the same syntax for executing them.

**Description**

The method Runtime.exec() should not be used to execute commands because the format of command execution is platform dependent.

**Example**

The following invocation of the exec() method would be flagged as a violation:

```
Runtime.exec("c:\\bin\\myApp.exe", new String[0]);
```

**Command Injection**

**Severity:** High

**Summary**

Commands being executed might be receiving data from the user or other unsafe sources.

**Description**

Command Injection occurs when the user is able to enter data directly into a command to be executed by the system.

The difference between Code Injection and Command Injection is that Code Injection attempts try to introduce scripts that are then included as part of the application runtime, while Command Injection attacks execute the user input source from outside of the application runtime.

To detect violations, this audit rule searches the code for command executions such as `java.lang.Runtime.exec(..)` and traces where the query data could have come from. In cases where the source of the command is user input, such as data from a servlet request, `javax.servlet.ServletRequest.getParameter(java.lang.String)`, or from a SWT Text widget, `org.eclipse.swt.widgets.Text.getText()`, a violation is created.

These two sets of methods, the locations where tainted user data can come from and the methods used to execute methods, are editable by the user.If methods are missing that are in a common package (such as `java.lang.*`), please let CodePro support know.

**Security Implications**

Successful Command Injection attacks can potentially have malicious commands executed with permissions not guarded against by the program.

**Example**

The invocation of the method `executeQuery(..)` would be flagged as a violation since it uses the first name information passed from a servlet request:

```
Text text = ...;
String command = text.getText();
...
Runtime.getRuntime().exec(command);
```

**Comment Local Variables**

**Severity:** Medium

**Summary**

Local variables should be commented.

**Description**

This audit rule checks for local variable declarations that are not followed by an end-of-line style comment. The comment is intended to describe the purpose of the variable.

If there are multiple local variables declared on a single line, the comment (if there is one) is assumed to belong to the first variable.

**Example**

The following local variable declaration would be flagged because it is not commented:

```
int sum = 0;
```

**Comparison Of Constants**

**Severity:** Medium

**Summary**

Constants should not be directly compared.

**Description**

Comparisons of two constant values waste processor cycles.

**Example**

Given the following declarations:

```
static final int ZERO = 0;
static final int ONE = 1;
```

The following condition would be flagged:

```
if (ZERO != ONE) {
```

**Comparison Of Incompatible Types**

**Severity:** Medium

**Summary**

Avoid comparing objects whose types do not have any common parent classes or interfaces using the `equals` method.

**Description**

This audit rule looks for places where incompatible types are compared using the `equals` method. Such a comparison will always return `false`, and thus usually indicates an error.

**Example**

The following invocation of the `equals` method will be marked as a violation because the types are incompatible:

```
public class MyClass {
.........
}

public void myMethod(MyClass obj, String a[]) {
if(obj.equals(a))
return;
}
}
```

**Comparison Of Short And Char**

**Severity:** Medium

**Summary**

Values of type short and char should not be directly compared.

**Description**

Comparisons between short and char values are performed by widening both to the type int and then performing the comparison. However, because shorts are signed and chars are unsigned, this can produce unintended results.

**Example**

The following would be flagged:

```
short s;
char c;
if (s == c) ...
```

**Complex Type Element Naming Convention**

**Severity:** Medium

**Summary**

Use upper camel case when naming elements that are complex types.

**Description**

This audit rule looks for declarations of complex type names that are not using upper camel case.

**Example**

The following element declaration has a complex type but its name is declared in lower camel case and would thus be marked as a violation:

```
<element
name="firstName"
type="StringTest"/>
```

**Complex Type Naming Convention**

**Severity:** Medium

**Summary**

Complex type names should conform to a naming standard.

**Description**

The following standard is being set to clarify definitions of complex types:
* Upper camel case should be used when naming complex types.
* When naming complex types, you must append 'Type' to the end of the name.
* Separate schemas/complex types for lists should not be created. Using the word 'list' in a schema causes confusion in the underlying implementation. Set maxOccurs="unbounded" to create a list of elements; this allows the bindings to determine the implementation details of a list structure.
* Element names must not be plural.

**Example**

The following complex type declarations have names that do not conform to a standard and would thus be marked as violations:

```
<complexType name="Book"/>
<complexType name="bookType"/>
```

They should be renamed to look something like the following:

```
<complexType name="BookType"/>
```

The following complex type declaration was only created to wrap a list and would thus be marked as a violation:

```
<complexType name="RetrieveCriteriaList">
<sequence>
<element
name="RetrieveCriteria"
type="retrieveCriteria:RetrieveKeyType"
maxOccurs="unbounded"/>
</sequence>
</complexType>
```

**Concatenation In Appending Method**

**Severity:** Medium

**Summary**

The argument to methods that append to buffers and streams.

**Description**

This audit rule finds places where the argument to a method that appends to a buffer or stream is the result of a concatenation. In such cases, a separate StringBuffer is being allocated to implement the concatenation, resulting an unnecessary overhead. The items that are being concatenated should be appended separately.

**Example**

The following invocation of the append method would be flagged as a violation:

```
buffer.append("Hello " + userName + "!");
```

**Concurrent Modification**

**Severity:** Medium

**Summary**

Removing from the collection being iterated over and continuing iteration afterwards results in an exception.

**Description**

This audit rule violates the cases when elements are removed from a collection while still iterating over it. Precisely, calling `Iterator#next()` after `remove()` was called results in an exception. Correct way of dealing with a necessity to do so is to iterate over the duplicate of a collection while removing found objects from the original or to use the iterators `remove()` method if it is known to be supported.

**Security Implications**

For fail-fast collections concurrent modifications result in `ConcurrentModificationException`, which could create an unexpected situation in the code execution and thus be a potential security threat.

**Example**

The following method removes elements from a collection while iterating over it and thus would be marked as violation:

```
public void removeByPattern(Collection c, String pattern) {
for (Iterator i = c.iterator(); i.hasNext();) {
String val = (String) i.next();
if (val.contains(pattern)) {
c.remove(val);
}
}
}
```

**Conditional Operator Use**

**Severity:** Low

**Summary**

The conditional operator should not be used.

**Description**

This audit rule finds places where the conditional operator (?:) has been used.

**Example**

The following use of the conditional operator would be flagged as a violation:

```
netSales = customerGetsDiscount(grossSales) ? computeDiscount(grossSales) :
grossSales;
```

**Configure Logging In File**

**Severity:** Medium

**Summary**

Logging should be configured in a file.

**Description**

This rule find places where a Configurator is used to programatically configure a logger. Configuration of loggers should be done by including configuration information in a configuration file in order to make it easier to change.

**Example**

The following invocation would be flagged as a violation:

```
configurator.doConfigure(null, null);
```

**Consistent Suffix Usage**

**Severity:** Medium

**Summary**

Type names should end with "Exception" if, and only if, they are derived from java.lang.Exception. Similarly, type names should end with "Error" if, and only if, they are derived from java.lang.Error.

**Description**

This audit rule finds types whose name ends with "Exception" but are not derived from

java.lang.Exception, or classes that are derived from java.lang.Exception but whose name does not end in "Exception".

**Example**

The following class declaration would be flagged as a violation because the class name ends with "Exception" but it not derived from the class java.lang.Exception:

```
public class RequiredClassException
{
...
}
```

The following class declaration would be flagged as a violation because it is derived from the class java.lang.Exception but does not end with "Exception":

```
public class InvalidAuthorizationCode extends Exception
{
...
}
```

**Constant Condition**

**Severity:** Medium

**Summary**

A constant expression in a conditional statement indicates either dead or debugging code.

**Description**

Using a constant as a conditional expression could indicate either debugging code or dead code.

**Security Implications**

Both variants are a potential security risk and should be avoided.

**Example**

The following statement was inserted as a way to skip a block of business code below it and should be removed:

```
if (true) return null;
```

**Constant Conditional Expression**

**Severity:** Medium

**Summary**

Conditional expressions should usually not be constant valued.

**Description**

This audit rule looks for conditional expressions in if, do, for, and while statements whose value is a compile-time constant. Because the value of such conditions cannot change, either the conditional code will never execute or will always execute (and in the case of a loop, the loop will never terminate).

**Example**

The expression in the following code would be flagged as a violation:

```
if (false) {
thisWillNeverBeExecuted();
}
```

**Constant Field Naming Convention**

**Severity:** Medium

**Summary**

Constant names should conform to the defined standard.

**Description**

This audit rule checks the names of all static fields that are also final.

**Example**

If constant fields are specified as needing to consist only of upper-case letters and the underscore, the following declaration would be flagged as a violation:

```
public static final int DefaultCount = 0;
```

**Constants in Comparison**

**Severity:** Medium

**Summary**

Constants should appear on the same side in comparisons.

**Description**

This audit rule looks for comparisons involving exactly one constant value and ensures that the constant appears on the side selected by the user. This is primarily useful to ensure consistency for ease of comprehension. There is some justification for specifying that constants should appear on the left because then the compiler will catch cases where the assignment operator was mistakenly used when a comparison operator was intended.

**Example**

If the rule has been configured so that constants are required to be on the left hand side, the following comparison would be flagged as a violation:

```
if (count > 0) {
...
}
```

**Constructors Only Invoke Final Methods**

**Severity:** Medium

**Summary**

Constructors should only invoke final methods on the object being constructed.

**Description**

Subclasses can override non-final methods. Invoking them from a constructor can cause errors because the object is not in a valid state.

**Example**

The constructor in the following class would be flagged as a violation:

```
public class Point
{
...
public Point()
{
x = initialX();
y = initialY();
}
protected int initialX()
{
return 0;
}
...
}
```

**Container Should Not Contain Itself As Element**

**Severity:** Medium

**Summary**

Re-adding objects in a container to its contents is usually a typo which could result in an inpredicted behaviour of a code.

**Description**

This audit rule violates container storing operations such as `addAll()` or `removeAll()` when they are invoked with the same container as an argument.

**Security Implications**

Such invocation is usually a typo which indicates a plain error in a logic of the code. Such code will not function as expected and could result in any security threat from Denial of Service to data leaks when used in security-sensitive areas.

**Example**

The following method is supposed to remove all banned users from the given list of users trying to access secure data but will fail because of the typo; this typo would thus be marked as violation:

```
protected void filterBannedUsers(List allUsers) {
List bannedUsers = new ArrayList();
for (Iterator i = allUsers.iterator(); i.hasNext();) {
User user = (User) i.next();
if (isBanned(user)) {
```

```
bannedUsers.add(user);
}
}
}
```

**Continue with Label**

**Severity:** Medium

**Summary**

Continue statements should not reference a labeled statement.

**Description**

This audit rule finds continue statements that reference a labeled statement. The use of a label with a continue statement makes the code much harder to read and maintain, and should therefore be avoided.

**Example**

The following continue statement would be flagged as a violation:

```
outer: for (int i = 0; i < array.length; i++) {
for (int j = 0; j < array[i].length; j++) {
if (array[i][j] == 0) {
continue outer;
}
...
}
}
```

**Convert Class to Interface**

**Severity:** Medium

**Summary**

Some classes could be converted to interfaces.

**Description**

This audit rule finds classes containing no methods or only abstract methods, and no fields or

only static final fields. Classes such as these could be converted to an equivalent interface providing increased implementation flexibility.

**Example**

The following class declaration would be flagged as a violation:

```
public abstract class RunnableWithException
{
public abstract void run()
throws Exception;
}
```

**Create Global Forward**

**Severity:** Medium

**Summary**

Use global forwards to avoid redundant forwards.

**Description**

This audit rule looks for ActionForwarder instances that refer to the same page. If there are several actions doing redirection to the same page, it is better to create and use a global redirect.

**Example**

Replace code like the following:

```
public class Action1 extends Action {
...
public ActionForward execute(...) throws Exception {
...
ActionForward forward = new ActionForward("/jsp/example.jsp");
return forward;
}
}
```

and

```
public class Action2 extends Action {
...
public ActionForward execute(...) throws Exception {
...
```

```
ActionForward forward = new ActionForward("/jsp/example.jsp");
return forward;
}
}
```

with code like the following:

```
public class Action1 extends Action {
...
public ActionForward execute(...) throws Exception {
...
return mapping.findForward("globalForwardName");
}
}
```

and

```
public class Action2 extends Action {
...
public ActionForward execute(...) throws Exception {
...
return mapping.findForward("globalForwardName");
}
}
```

Of course, you must also define your global forward in the configuration file, something like the following:

```
<global-forwards>
<forward name="globalForwardName" path="/jsp/example.jsp" />
</global-forwards>
```

**Cross-Site Scripting**

**Severity:** High

**Summary**

User input might be getting printed directly out to a web site.

**Description**

Cross-Site Scripting occurs when the user is able to enter data directly onto a web site they are visiting.

To detect violations, this audit rule searches the code for printing statements such as `HTTPServletResponse.getWriter().print(..)` and traces where the output data could have come from. In cases where the source of the write statement is user input, such as data from a

servlet request, `javax.servlet.ServletRequest.getParameter(java.lang.String)`, or from a SWT Text widget, `org.eclipse.swt.widgets.Text.getText()`, a violation is created.

These two sets of methods, the locations where tainted user data can come from and the methods used to potentially print to a web site, are editable by the user. If methods are missing that are in a common package (such as `java.lang.*`), please let CodePro support know.

**Security Implications**

If a malicious user has access to enter a piece of JavaScript, the next user to load the web page could potentially have their cookies or other resources compromised.

**Example**

The invocation of the method `print(..)` would be flagged as a violation since it prints the first name information passed from a servlet request:

```
ServletRequest servletRequest = ...;
HttpServletResponse httpServletResponse = ...;

String firstName = servletRequest.getParameter("firstName");

httpServletResponse.getWriter().print(firstName);
```

**Cyclomatic Complexity**

**Severity:** Low

**Summary**

Methods should be kept fairly simple.

**Description**

This audit rule finds methods, constructors and initializers that are too complex. The complexity is measured by the number of "if", "while", "do", "for", "?:", "catch" and "switch" statements (plus one) in the body of the member.

**Example**

If the maximum cyclomatic complexity has been configured to be 3, then the following method would be flagged as a violation because it has a cyclomatic complexity of 4:

```
public int getHireYear()
{
EmploymentRecord employmentRecord = getEmploymentRecord();
if (employmentRecord == null) return 0;
HiringRecord hiringRecord = employmentRecord.getHiringRecord();
if (hiringRecord == null) return 0;
Calendar hireDate = hiringRecord.getHireDate();
if (hireDate == null) return 0;
return hireDate.getYear();
}
```

**Dangling Else**

**Severity:** Medium

**Summary**

Use blocks to prevent dangling else clauses.

**Description**

This audit rule finds places in the code where else clauses are not preceded by a block because these can lead to dangling else errors.

**Example**

```
if (a > 0)
if (a > 100)
b = a - 100;
else
b = -a;
```

**Database Connections Should Not Be Static**

**Severity:** Medium

**Summary**

A database connection shared via a `static` field can be accessed simultaneously by multiple threads. This is against the transaction-based nature of a `Connection` resource.

**Description**

`JDBC Connection` is a transactional resource object. Such objects can only be associated with one transaction at a time.

**Security Implications**

Storing connections in static fields would make it easy to erroneously share them between threads in different transactions, thus creating a potentially dangerous situation for the data.

**Example**

The following code would be flagged as a violation because it shares a JDBC Connection via static field:

```
public class DataRequest {
private final static Connection connection;
...
}
```

**Date and Time Usage**

**Severity:** Low

**Summary**

Don't use deprecated Date and Time methods.

**Description**

Deprecated methods from the Date and Time classes should not be used. As of JDK 1.1, the Calendar class should be used to convert between dates and time fields, and the DateFormat class should be used to format and parse date strings.

**Example**

The following use of the parseInteger() method would be flagged as a violation:

```
Date myDate = Date(2005, 10, 12);
int year = myDate.getYear();
```

**Debugging Code**

**Severity:** Medium

**Summary**

Debugging and profiling code should not be left in production code.

**Description**

This audit rule finds places in the code where text is being written to either System.out or System.err, or where the methods Throwable.printStackTrace(), Thread.dumpStack(), Runtime.freeMemory(), Runtime.totalMemory(), Runtime.traceMethodCalls() or Runtime.traceInstructions() are being invoked.

**Example**

```
try {
...
} catch (Exception exception) {
System.err.println("Unexpected exception:");
exception.printStackTrace();
}
```

**Declare Accessors for All ActionForm Fields**

**Severity:** High

**Summary**

Declare public accessors for all ActionForm fields.

**Description**

This audit rule looks for fields declared in ActionForm classes for which either the getter or setter method is not implemented. If accessor methods are not implemented, using this tag on a JSP page such as

```
<html:text property="fieldName">
```

may cause an error.

**Example**

The following class would be flagged because it does not implement a setter method for the "username" field.

```
public class LoginForm extends ActionForm {
private String password = null;
private String username = null;

public String getUsername() {
return (this.username);
}

public String getPassword() {
return (this.password);
}

public void setPassword(String password) {
this.password = password;
}
}
```

**Declare Accessors for Persistent Fields**

**Severity:** Medium

**Summary**

Declare accessor methods for all persistent fields of persistent class.

**Description**

This audit rule looks for fields in persistent classes for which either the getter or setter method is missing. Many other ORM tools directly persist instance variables, but Hibernate decouples this implementation detail from the persistence mechanism. Hibernate persists JavaBeans style properties, and recognizes method names of the form getFoo, isFoo and setFoo. Properties need not be declared public - Hibernate can persist a property with a default, protected or private get/set pair.

**Example**

The fields "text" and "nextMessage" would be flagged because they do not have getter or setter methods.

```
public class Message {
private Long id;
private String text;
public Message nextMessage;
```

```
Message() {}

public Message(String text) {
this.text = text;
}

public Long getId() {
return id;
}

private void setId(Long id) {
this.id = id;
}

public String getText() {
return text;
}
}
```

**Declare As Interface**

**Severity:** Medium

**Summary**

Variables of certain types should be declared using an interface.

**Description**

This audit rule finds declarations of fields, local variables, and methods whose type should have been declared to be an interface but was declared to be a class that implements the interface instead. The list of interfaces that are checked can be configured.

**Example**

If the type java.util.List is on the list of interfaces, the following would be flagged as a violation because the declared type of the field should have been "List":

```
private ArrayList myList;
```

**Declare Default Constructors**

**Severity:** Medium

**Summary**

Types should declare a default constructor.

**Description**

This audit rule finds class declarations that do not contain the declaration of a default (zero argument) constructor.

**Example**

The following class declaration would be flagged as a violation because it does not implement a default constructor:

```
public class Employee
{
public Employee(String name)
{
...
}
}
```

**Declare Identifier Properties**

**Severity:** Medium

**Summary**

Declare identifier properties on persistent classes.

**Description**

This audit rule looks for persistent classes that do not have an identifier property declared for them. Although Hibernate makes identifier properties optional, some functionality is available only to classes which declare an identifier property, including

1. Transitive reattachment for detached objects (cascade update or cascade merge)
2. Session.saveOrUpdate()
3. Session.merge()

**Example**

The following class would be flagged because there is no identifier field declared for it.

```
public class Message {
private String text;
private Message nextMessage;

Message() {}

public Message(String text) {
this.text = text;
}

public String getText() {
return text;
}

public void setText(String text) {
this.text = text;
}

public Message getNextMessage() {
return nextMessage;
}

public void setNextMessage(Message nextMessage) {
this.nextMessage = nextMessage;
}
}
```

**Declare Private Identifier Setter**

**Severity:** Low

**Summary**

The setter method for an identifier property should be private.

**Description**

This audit rule looks for non-private methods used to set the value of an identifier field. The identifier field should only be assigned a value by Hibernate when the object is saved. By making the method private you ensure that other code cannot change the object's identity.

**Example**

The "setId" method would be flagged because it should be private.

```
public class Message {
private Long id;
private String text;
private Message nextMessage;
```

```
Message() {}

public Message(String text) {
this.text = text;
}

public Long getId() {
return id;
}

public void setId(Long id) {
his.id = id;
}
}
```

**Declare Setters for Bean Fields**

**Severity:** Medium

**Summary**

Declare setter methods for all fields of bean class.

**Description**

This audit rule looks for fields declared in beans for which no setter methods are declared. Setter injection is the preferred type of dependency injection in Spring. Setter injection is more flexible and manageable. Your bean classes should have setters for all fields to use this technique.

**Example**

The field "text" would be flagged because it does not have a setter method:

```
public class Bean {

private String text;

public String getText() {
return text;
}

...
}
```

**Declare Type for java.util.Date Property**

**Severity:** Medium

**Summary**

Declare type for java.util.Date property in configuration file.

**Description**

This audit rule looks for declarations of properties in the configuration files whose corresponding field has the type java.util.Date, but for which the property file does not contain a "type" attribute. Without a "type" attribute, Hibernate can't know if the property should map to an SQL date, a timestamp, or a time column.

**Example**

Given a class defined like the following

```
public class Message {
private Long id;
private String text;
private Date date;

public Message () {}

public Long getId() {
return id;
}

private void setId(Long id) {
this.id = id;
}

...
}
```

and a corresponding configuration file containing the following

```
<hibernate-mapping>
<class name="messages.Message" table="MESSAGES">
<id name="id" column="MESSAGE_ID">
<generator class="native"/>
</id>
<property name="date" column="MESSAGE_DATE"/>
<property name="text"/>
</class>
...
</hibernate-mapping>
```

The property "date" would be flagged because it does not include the attribute "type". It should be replaced by something like the following

```
<property name="date" type="timestamp" column="MESSAGE_DATE"/>
```

**Declared Exceptions**

**Severity:** Medium

**Summary**

Some exceptions should not be declared for methods or constructors.

**Description**

This audit rule finds methods and constructors that declare as a thrown exception an exception class that is disallowed. The list initially includes exception classes that are either too general (such as Throwable or Exception), or that are unchecked (Error, RuntimeException, and all subclasses of either).

**Example**

If the rule is configured to disallow the declaration of unchecked exceptions, then the following method would be flagged as a violation:

```
public void initialize()
throws Error
{
...
}
```

**Default Namespace**

**Severity:** Medium

**Summary**

The default namespace xmlns should be set without using a prefix.

**Description**

This audit rule looks for schema xmlns declarations that refer to the http://www.w3.org/2001/XMLSchema URI using a prefix such as "xsd:". Not using a prefix removes the need to have the prefix for all schema components such as element, complexType and built in types.

**Example**

The following schema reference would be marked as a violation:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" ... >
```

Use the following instead:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" ... >
```

**Default Not Last in Switch**

**Severity:** Medium

**Summary**

The default case should be the last case in a switch statement.

**Description**

This audit rule finds switch statements in which the default case is not the last case. Making the default case always be last improves the readability of the code by making it easier to locate the default behavior.

**Example**

The following case statement would be flagged as a violation because the default case is not the last case:

```
switch (registrationType) {
case 0: \ audit
...
default:
case 1: \ for credit
...
case 2: \ pass/fail
...
}
```

**Define Constants in Interfaces**

**Severity:** Medium

**Summary**

Constants should be defined in an interface.

**Description**

This audit rule checks for constants (public static final fields) that are declared in a class. Such fields should be declared in an interface so that they are easier to share and can be referenced without introducing extra coupling.

**Example**

The following field declaration would be flagged if it occurred outside of an interface:

```
public static final int DAYS_IN_WEEK = 7;
```

**Define Initial Capacity**

**Severity:** Low

**Summary**

Define an initial capacity for ArrayList, HashMap, HashSet, Hashtable, StringBuffer, StringBuilder, WeakHashMap, or Vector instances.

**Description**

This audit rule requires an initial capacity to be specified when creating instances of ArrayList, HashMap, HashSet, Hashtable, StringBuffer, StringBuilder, WeakHashMap, or Vector. Array capacity expansion involves allocating a larger array and copying the contents of the old array to a new one. The old array object eventually gets reclaimed by the garbage collector. Array expansion is an expensive operation which should be avoided where possible. If you are able to approximate the expected size, you should use this value instead of the default.

**Example**

The following instance creation would be flagged as a violation because the expected size of the collection is not specified:

```
new ArrayList();
```

**Define Load Factor**

**Severity:** Low

**Summary**

Define the load factor when creating instances of the classes 'HashMap', 'HashSet', 'Hashtable', and 'WeakHashMap'.

**Description**

This audit rule requires a load factor to be specified when creating instances of the classes 'HashMap', 'HashSet', 'Hashtable', and 'WeakHashMap'. The load factor impacts both the size of the collection and how long it will take to perform a look-up.

**Example**

The following instance creation would be flagged as a violation because the load factor is not specified:

```
new HashMap(42);
```

**Delete Temporary Files**

**Severity:** Medium

**Summary**

If a file contains sensitive information, it is better to delete it as soon as possible.

**Description**

If a file contains sensitive information, it is better to delete it as soon as possible. This audit rule looks for places where a file created using `File.createTempFile()` and not deleted explicitly with the `delete()` method before the method returns.

**Security Implications**

Using the method `deleteOnExit()` is not enough because, especially in web development, an application can run for a significantly long time. We also assume that storing sensitive data for the duration of a session is also insecure.

**Example**

The following code would be flagged as a violation because it does not delete the created temporary file:

```
public void doSomeStoring() {
File temp = File.createTempFile();
...
}
```

**Denial of Service Threat**

**Severity:** High

**Summary**

Request parameters and other tainted data should not be passed into methods that could assist the Denial of Service state without sanitizing.

**Description**

This rule violates usage of an unvalidated user input in calls to methods that could assist the Denial of Service state.

**Security Implications**

Only trusted data should be used in such methods, otherwise an attacker can provide data that will lead to fast depletion of a thread pool or other Denial of Service states.

**Example**

The following code uses user input directly in a call to `Thread.sleep()`. By providing large values, an attacker can suspend a thread execution for a significant amount of time, which will ease the depletion of a thread pool:

```
String pause = req.getParameter("pause");
int seconds = Integer.valueOf(pause).intValue();
Thread.sleep(seconds * 1000);
```

**Deploy Mappings With Mapped Classes**

**Severity:** Medium

**Summary**

Deploy the mapping files along with the classes they map.

**Description**

This audit rule looks for mapping files that are not in the same directory as the class being mapped. Having the mapping documents in a different directory makes it harder to keep the information in the file up to date and makes it harder to refactor the code when necessary. For example, the class com.eg.Foo should be mapped in the file com/eg/Foo.hbm.xml. This makes particularly good sense in a team environment.

**Example**

Given the following entry in the "hibernate.cfg.xml" file

```
<mapping resource="helloMessage.hbm.xml"/>
```

The file "helloMessage.hbm.xml", containing the mapping entry below, would be flagged because it is not in the same directory as the mapped resource.

```
<hibernate-mapping>
<class
name="hello.Message"
table="MESSAGES">

...

</class>
</hibernate-mapping>
```

**Deprecated Method Found**

**Severity:** Medium

**Summary**

Deprecated API is error-prone and is a potential security threat and thus should not be used.

**Description**

Old API is sometimes marked deprecated because its implementation is designed in a way that can be error-prone. Deprecated API should be avoided where possible.

**Security Implications**

Blocks of code that use deprecated API are designed in a careless manner and thus are a potential security threat.

**Example**

The following code would be flagged as a violation because it uses a deprecated method:

```
public void resumeChild() {
getChildThread().resume();
}
}
```

**Dereferencing Null Pointer**

**Severity:** High

**Summary**

A null value is dereferenced as if it had an object value.

**Description**

Method invocation and field access on null values will cause errors. In expressions following explicit comparisons to null, such as (x == null && x.getValue()), dereferencing the null pointer will cause an error. These usually indicate typographical errors such as substituting && for || in the boolean expression.

**Example**

The method invocation in the following condition would be flagged as a violation:

```
if (object == null && object.getData() == null) ...
```

**Deserializeability Security**

**Severity:** Medium

**Summary**

Prevent non-deserializeable classes from being deserialized by adversaries.

**Description**

This audit rule enforces classes to implement a readObject method:

```
private final void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException {
throw new IOException("Non-deserializable class");
}
```

This audit will be violated if there does not exist a method with the signature:

```
private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException;
```

**Security Implications**

Even when a class is not serializeable, adversaries can sometimes instantiate an instance of a class with a byte array. This can be a security hazard since there is no control on the state of the new object.

**Example**

The following method declaration would be flagged as a violation because it does not have the required parameter:

```
private void readObject() throws IOException, ClassNotFoundException;
```

**Detect Multiple Iterations**

**Severity:** Low

**Summary**

Detect multiple iterations over a single collection.

**Description**

A single method that contains multiple loops that iterate over the contents of a single collection

may be improved by rewriting it. If it is possible to use a single loop then the loop maintenance overhead can be eliminated.

**Example**

The second for loop would be flagged as a violation:

```
for (int i = 0; i < array.length; i++) {
...
}
...
for (int i = 0; i < array.length; i++) {
...
}
```

**Disallow @Test Annotation**

**Severity:** Medium

**Summary**

The @Test annotation should not be used.

**Description**

This audit rule finds uses of the @Test annotation.

**Example**

The following annotation would be flagged as a violation:

```
@Test
public void testIt()
```

**Disallow Array Initializers**

**Severity:** Medium

**Summary**

Arrays should not be statically initialized by an array initializer.

**Description**

This audit rule checks for array variables that are initialized (either in the initializer or in an assignment statement) using an array initializer.

**Example**

The following array declaration would be flagged because of the use of an array initializer:

```
int[] values = {0, 1, 2};
```

**Disallow AST toString()**

**Severity:** Medium

**Summary**

The method toString() should never be invoked for AST nodes.

**Description**

This audit rule looks for invocations of the method toString() on AST nodes. The toString() method produces a string that should only be used for debugging purposes, so it is almost never the right method to use.

**Example**

Given the following declaration:

```
SimpleName name;
```

The following invocation would be flagged:

```
name.toString()
```

**Disallow Default Package**

**Severity:** Low

**Summary**

Code should not be defined in the default package.

**Description**

This audit rule looks for compilation units that are declared in the default package. All code should be structured into packages.

**Disallow Native Methods**

**Severity:** Medium

**Summary**

Native methods should be avoided because they are often platform dependent.

**Description**

Native methods should be avoided because they are often platform dependent.

**Security Implications**

When native methods are executed, the execution path leaves the Java API, leaving behind all Java security features, including any setup security managers.

**Example**

The following method declaration would be flagged as a violation because it is implemented as a native method:

```
public int native getUID();
```

**Disallow Notify Usage**

**Severity:** Medium

**Summary**

The notify() method should not be used.

**Description**

The method notifyAll() should be used rather than notify() because it will generally lead to a better scheduling of waiting threads.

**Example**

The following invocation of the notify() method would be flagged as a violation:

```
synchronize (monitor) {
monitor.notify();
}
```

**Disallow Sleep Inside While**

**Severity:** Medium

**Summary**

The sleep() method should not be used within a while loop.

**Description**

This audit rule looks for invocations of the sleep() method that occur inside of a while loop. Such occurrences usually indicate that the code implements a busy-wait loop, which is inefficient. Instead, the code should use wait() and notify() to block the thread until it is possible for execution to proceed.

**Example**

The following invocation of the sleep() method would be flagged as a violation:

```
while (eventQueue.isEmpty()) {
Thread.sleep();
}
```

**Disallow Sleep Usage**

**Severity:** High

**Summary**

The sleep() method should not be used.

**Description**

This audit rule looks for invocations of the method Thread.sleep(). If you are waiting for some specific state to be reached, you should instead use the wait() method so that you can be notified when the state has been reached. This will improve both the performance and reliability of your code.

**Example**

The following invocation of the sleep() method would be flagged as a violation:

```
sleep(1000);
```

**Disallow Temporary Sessions**

**Severity:** Medium

**Summary**

JSP pages should not allow temporary sessions.

**Description**

This audit rule checks for JSP files that do not contain a page directive that explicitly disallows temporary sessions.

**Disallow ThreadGroup Usage**

**Severity:** Medium

**Summary**

The class ThreadGroup should not be used.

**Description**

The class ThreadGroup should not be used in multi-threaded applications because its implementation is not thread safe.

**Example**

The following instance creation would be flagged as a violation because it is creating an instance of the class java.lang.ThreadGroup:

```
new ThreadGroup("Background Threads")
```

**Disallow Unnamed Thread Usage**

**Severity:** Medium

**Summary**

Threads should be named to aid in debugging.

**Description**

This audit rule looks for the creation of threads that are not given a name. Threads should be named in order to make it easier to debug the application.

**Example**

The following thread creation would be flagged as a violation because a name is not provided:

```
new Thread(new Runnable() { ... });
```

**Disallow Use of AWT Peer Classes**

**Severity:** High

**Summary**

AWT peer classes should not be referenced because they are platform dependent.

**Description**

The AWT peer classes provide a platform-specific implementation of the AWT widgets. Referencing any of these classes directly will result in platform dependent code.

**Example**

The following reference to the class sun.awt.windows.WButtonPeer would be flagged as a violation because it is specific to the Windows platform:

```
if (peer instanceof sun.awt.windows.WButtonPeer)
```

**Disallow Use of Deprecated Thread Methods**

**Severity:** Medium

**Summary**

Don't use deprecated methods when writing multi-threaded code.

**Description**

The methods Thread.resume(), Thread.stop(), Thread.suspend(), Thread.destroy() and Runtime.runFinalizersOnExit() have been deprecated and should not be used because they are inherently unsafe.

**Example**

The following use of the suspend() method would be flagged as a violation:

```
processingThread.suspend();
```

**Disallow Yield Usage**

**Severity:** Medium

**Summary**

The method Thread.yield() should not be used.

**Description**

The method Thread.yield() should not be used because its behavior is not consistent across all platforms.

**Example**

The following invocation of the yield() method would be flagged as a violation:

```
backgroundTask.yield();
```

**Disallowed File**

**Severity:** Medium

**Summary**

Some files should not exist in a project.

**Description**

This audit rule finds files within projects that should be deleted. The list of files is configurable, but typically includes auxilliary files created by utility programs.

**Dispose Should Be Invoked**

**Severity:** Medium

**Summary**

You should invoke the `dispose()` method for objects which are types or subtypes of classes that are situated in `org.eclipse.swt.graphics` package.

**Description**

This rule looks for places where such objects are created and then checks that `dispose()` was invoked.

**Security Implications**

An SWT resource that is not properly closed is a reasource leak. When created, an SWT object takes system resources that will not be freed if dispose() is not called. Sooner or later this will cause denial in creation of new SWT objects resulting in a runtime exception. This could be used to create a potential denial-of-service state or reveal security-sensitive parts of an application's design through the stack trace.

**Example**

The following creation of an object would be marked as a violation because the `dispose()` method is not invoked:

```
public class Sample {
public void sampleMeth( Display display ) {
ImageData imageData = new ImageData("filename");
Image image = new Image(display, imageData);
}
}
```

**Do not Access/Modify Security Configuration Objects**

**Severity:** Medium

**Summary**

Security policy should not be accessed or modified from EJB beans.

**Description**

This audit rule looks for code that accesses or modifies security objects in an EJB context.

**Security Implications**

The security of an application on the JVM execution level is implemented on a web application container level. Trying to manipulate it from inside the EJB code indicates a potentially dangerous approach to security of an application. An application should be reworked so that such access to security data is not required.

**Example**

The following code would be flagged as a violation because it modifies a security object:

```
public class ClassOne implements EntityBean {
public void access (){
Policy.setPolicy(new MyPolicy());
}
....
}
```

**Do not Catch IllegalMonitorStateException**

**Severity:** Medium

**Summary**

Catching an IllegalMonitorStateException indicates a concurrency handling error in debug.

**Description**

This audit rule looks for code that attempts to catch `IllegalMonitorStateException`.

**Security Implications**

Catching an `IllegalMonitorStateException` indicates a concurrency handing flaw in the design of an application which is only partially mitigated, but not avoided. This design flaw could be exploited to bring an application to a denial of service state or even provide security-sensitive data, such as parts of an application design, to the attacker.

**Example**

The following code would be flagged as a violation because it catches `IllegalMonitorStateException`:

```
public void myMethod() {
try {
//...
} catch (IllegalMonitorStateException e ) {
//...
}
}
```

**Do Not Create Finalizable Objects**

**Severity:** Medium

**Summary**

Finalizable objects should not be instantiated.

**Description**

This audit rule checks for instance creation expressions in which the object being created implements the finalize() method. Finalization is expensive and error prone, so finalizable objects should not be used.

**Example**

If the class ResourceHandle defines the finalize method, then the following instance creation expression would be flagged as a violation:

```
ResourceHandle handle = new ResourceHandle(resourceId);
```

**Do Not Declare Bindings**

**Severity:** Medium

**Summary**

Binding information should not be included in XML schemas.

**Description**

This audit rule looks for any usage of the `<wsdl:binding>` tag.

**Example**

The following WSDL definition declares bindings and would thus be marked as a violation:

```
<wsdl:definitions .... >
<wsdl:binding name="nmtoken" type="qname"> *
<!-- extensibility element (1) --> *
<wsdl:operation name="nmtoken"> *
<!-- extensibility element (2) --> *
<wsdl:input name="nmtoken"?> ?
<!-- extensibility element (3) -->
</wsdl:input>
<wsdl:output name="nmtoken"? >?
<!-- extensibility element (4) --> *
</wsdl:output>
<wsdl:fault name="nmtoken"> *
<!-- extensibility element (5) --> *
</wsdl:fault>
</wsdl:operation>
</wsdl:binding>
</wsdl:definitions>
```

**Do Not Implement Outdated Interfaces**

**Severity:** Medium

**Summary**

Outdated interfaces should not be implemented. Instead, implement the ones that superseded them

**Description**

This audit rule flags classes that implement outdated interface, or extend outdated abstract classes.

**Example**

The following would be flagged because Iterator could be implemented instead:

```
public class Foo extends Enumeration{}
```

**Do Not Implement Serializable**

**Severity:** Medium

**Summary**

Objects and interfaces should not be declared as Serializable.

**Description**

This audit rule violates classes and interfaces that implement Serializable.

**Security Implications**

When an object is serialized it is outside of the JRE and thus outside of all security measures enforced by the JRE. In many cases, packages or entire projects should not declare objects or interfaces as Serializable.

**Example**

The following class would be flagged:

```
class A implements java.io.Serializable ...
```

**Do Not Invoke setSize()**

**Severity:** High

**Summary**

The method Component.setSize() should not be invoked within ComponentListener.componentResized().

**Description**

This audit rule looks for invocations of the method Component.setSize() within methods that override the method ComponentListener.componentResized(). The method setSize() causes the method componentResized() to be invoked, thus leading to a stack overflow.

**Example**

The given invocation of the method setSize() in the following method would be flagged as a violation:

```
class MyComponentListener extends Component implements ComponentListener {
public void componentResized(ComponentEvent event) {
setSize(100, 100);
}
...
}
```

**Do Not Serialize Byte Arrays**

**Severity:** Medium

**Summary**

Do not serialize byte arrays using the ObjectOutputStream.write(byte[]) method.

**Description**

This audit rule violates invocations of the method ObjectOutputStream.write(byte[]).

**Security Implications**

Byte arrays should not be serialized using `write(byte[])` because the method is not final, allowing subclasses of ObjectOutputStream to override the method and get access to internal data.

**Example**

Given the following field declaration:

private byte[] key;

The following invocation of the write method would be flagged:

```
private void writeObject(ObjectOutputStream stream)
{
stream.write(key);
}
```

**Do Not Subclass ClassLoader**

**Severity:** Medium

**Summary**

Do not extend `java.lang.ClassLoader`

**Description**

Subclasses of `java.lang.ClassLoader` are flagged. In instances where `ClassLoader` needs to be subclassed, use `java.security.SecureClassLoader`.

**Security Implications**

By subclassing `SecureClassLoader` instead of `ClassLoader`, additional Java security measures are taken to protect against class loading vulnerabilities.

**Example**

The following class would be flagged as a violation:

```
class A extends ClassLoader {...}
```

**Document Closing Braces**

**Severity:** Medium

**Summary**

Closing braces should be documented with a comment.

**Description**

This audit rule checks for closing braces for specified statements that are not followed by a specified end-of-line comment. The comments are used to make it easier for readers to see the structure of the code.

**Example**

The closing brace for the following if statement would be flagged as needing to be commented:

```
if (employee.isHourly()) {
...
}
```

**Don't Create Unused Error**

**Severity:** Medium

**Summary**

Don't create ActionErrors in validate() methods of ActionForm until an error is discovered.

**Description**

This audit rule finds all created ActionError and ActionMessage instances and checks to see whether they are added (with method add) to the returned ActionErrors instance.

**Example**

In the following method, the error assigned to the local variable named "error" would be flagged because it not added to the "errors" collection:

```
public ActionErrors validate(ActionMapping mapping,
HttpServletRequest request) {
ActionError error = new ActionError(null);
ActionMessage message;
message = new ActionMessage(null);
ActionErrors errors = new ActionErrors();
errors.add(ActionErrors.GLOBAL_ERROR, new ActionError("error.error"));
errors.add(ActionErrors.GLOBAL_ERROR, message);
return errors;
}
```

**Don't Encode Markup in Renderer**

**Severity:** Medium

**Summary**

Hard-coding markup as Java literal strings and outputting using write() should be avoided.

**Description**

This rule looks for HTML/XML tags being output inside write() calls. Placing snippets of HTML code inside Java strings and outputting them using write() calls leads to less maintainable code and potential escaping problems, and is thus prohibited by some coding standards.

Two possible alternatives include:
- use startElement(), writeAttribute(), endElement() calls to output small amounts of markup, or
- use some templating solution to output larger amounts of markup.

**Example**

The following invocations of the write method would be flagged because the arguments contain markup.

```
public class CompARenderer extends Renderer {
public void encodeEnd(FacesContext context, UIComponent component)
throws IOException {
ResponseWriter writer = context.getResponseWriter();
writer.write("<div class="bpui_compA_popMid">");
writer.write("</div>");
}
}
```

**Don't Instantiate Beans**

**Severity:** Medium

**Summary**

The method instantiate() should not be used to instantiate beans.

**Description**

This audit rule looks for places where one of the instantiate() methods is being used to instantiate a bean.

**Example**

The following use of the instantiate() method would be flagged:

```
newBean = Beans.instantiate(classLoader, className);
```

**Don't Return Mutable Types**

**Severity:** Medium

**Summary**

Don't return mutable types from methods.

**Description**

Checks that returned types are either default immutable types like java.lang.Integer or are declared immutable in the audit rule preferences.

**Security Implications**

Methods that return sensitive data that is passed in a mutable form can potentially allow malicious users access to change the data.

**Example**

The following method declaration would be flagged (assuming "MutableType" is not declared as immutable in the audit rule preferences):

```
public MutableType getData() {
return new MutableType();
}
```

**Don't use concatenation to convert to String**

**Severity:** Medium

**Summary**

Don't use concatenation to convert to String.

**Description**

Concatenation with the empty string should never be used to convert something to a string. It is more efficient to use String.valueOf() to convert primitives, or toString() to convert objects.

**Example**

The following would be flagged as a violation:

```
count = 5; return count + "";
```

**Don't Use Default Bean Names**

**Severity:** Medium

**Summary**

Specify id as the bean identifier.

**Description**

This audit rule looks for places where the bean name is not explicitly given. If you give the tag an id attribute, then the value of that attribute is used as the name. If no id attribute is specified, Spring looks for a name attribute and, if one is defined, it uses the first name defined in the name attribute. (We say the first name because it is possible to define multiple names within the name attribute.) If neither the id nor the name attribute is specified, Spring uses the bean's class name as the name, provided, of course, that no other bean is using the same name.

Avoid using the automatic name by class behavior. This doesn't allow you much flexibility to define multiple beans of the same type, and it is much better to define your own names. That way, if Spring changes the default behavior in the future, your application will continue to work.

You can specify either an id or name as the bean identifier. Using ids will not increase readability, but it can leverage the XML parser to validate the bean references.

When choosing whether to use id or name, always use id to specify the bean's default name. The only drawback of using the id attribute is that you are limited to characters that are allowed within XML element IDs. If you find that you cannot use a character you want in your name, then you can specify that name using the name attribute, which does not have to adhere to the XML naming rules.

**Example**

Don't use

```
<bean name="string2" class="java.lang.String"/>
```

or

```
<bean class="java.lang.String"/>
```

Always use the following instead

```
<bean id="string1" class="java.lang.String"/>
```

**Don't use HTML Comments**

**Severity:** Medium

**Summary**

JSP pages should not use HTML comments.

**Description**

This audit rule looks for uses of HTML comments within JSP pages. HTML comments should not be used because they end up being sent to the client, increasing network traffic and potentially making internal implementation details visible.

**Example**

The following uses of an HTML comment would be flagged as a violation:

```
<!-- Backdoor hack -->
```

**Double Check Locking**

**Severity:** Medium

**Summary**

Double Check Locking singleton synchronization technique should not be used.

**Description**

Double-checked locking is a widespread practice used in Singleton pattern implementations. It is supposed to provide a safe way for the lazy initialization of the singleton in a multi-threaded environment combined with performance optimization. But in some cases this code will not work as intended, leading to concurrency problems and unpredictable behavior. This pattern should not be used.

**Security Implications**

in some cases double-checked locking will not work as intended, leading to concurrency problems and unpredictable behavior.

**Example**

The following code would be flagged as a violation because it uses the double-checked locking pattern:

```
public class Sample {
private List data;
public List getData() {
if (data == null) {
synchronized (this) {
if (data == null) {
data = new ArrayList();
}
}
}
return data;
}
}
```

**Duplicate Import Declarations**

**Severity:** Medium

**Summary**

Types and packages should only be imported once within a compilation unit.

**Description**

This audit rule checks to ensure that there are never more than one import declaration with the same package or type name. This does not flag the case where a package is imported (a demand import) and a type from the package is separately imported explicitly. (Such a situation is sometimes required to disambiguate names.)

**Example**

```
import java.util.Vector;
import java.util.Vector;
```

**Duplicate Property Name**

**Severity:** High

**Summary**

Properties should only be declared once.

**Description**

This audit rule checks to ensure that there is never more than one property declaration with the same name in the same file. Declaring multiple properties with the same name is usually a mistake because only the last such property will be visible to the code.

**Example**

```
applicationName = CoolApp
applicationName = Ice Machine Controller
```

**Duplicate Property Value**

**Severity:** Low

**Summary**

Properties should only be declared once.

**Description**

This audit rule checks to ensure that there is never more than one property declaration with the same value in the same file. Declaring multiple properties with the same value can needlessly increase the amount of effort required to translate the property values into another locale.

**Example**

```
importBottonLabel = Import...
importMenuLabel = Import...
```

**Duplicate Validation Form**

**Severity:** Medium

**Summary**

Multiple validation forms with the same name indicate that the validation logic is not up-to-date.

**Description**

If two validation forms have the same name, the Struts Validator arbitrarily chooses one of the forms to use for input validation and discards the other. This decision might not correspond to the programmer's expectations. Moreover, it indicates that the validation logic is not being maintained, and can indicate that other, more subtle, validation errors are present.

**Example**

If your validaton.xml file contains

```
<form-validation>
<formset>
<form name="ProjectForm">
...
</form>
<form name="ProjectForm">
```

```
...
</form>
</formset>
</form-validation>
```

then the second entry of form name="ProjectForm" would be flagged.

**Dynamic Dependency in Ivy**

**Severity:** Medium

**Summary**

Using a dynamic dependency version is a security risk.

**Description**

This audit rule violates the usage of dynamic dependency version in Ivy configuration files.

**Security Implications**

A dynamic dependency version adds to the number of undefined variables at the time of build that can be used by an attacker. More than that, you cannot validate the quality and security issues of the code used in the build. This is an additional security risk that should be taken into consideration.

**Example**

The following part of an Ivy script would be flagged as a violation because it declares a dependency with dynamically defined revision:

```
<dependency org="yourorg" name="yourmodule9" rev="9.1+" conf="A,B->default">
<include name="art1" type="jar" conf="A,B"/>
<include name="art2" type="jar" conf="A"/>
</dependency>
```

**Dynamic Dependency in Maven**

**Severity:** Medium

**Summary**

Using a dynamic dependency version is a security risk.

**Description**

This audit rule violates the usage of dynamic dependency version in Maven configuration files.

**Security Implications**

A dynamic dependency version adds to the number of undefined variables at the time of build that can be used by an attacker. More than that, you cannot validate the quality and security issues of the code used in the build. This is an additional security risk that should be taken into consideration.

**Example**

The following part of an Maven POM would be flagged as a violation because it declares a dependency with dynamically defined revision:

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>[,3.8.1]</version>
<scope>test</scope>
</dependency>
```

**Dynamically Compose Test Suites**

**Severity:** Medium

**Summary**

Test suites should be dynamically composed.

**Description**

This audit rule checks for test cases that implement the static method suite(). Implementing this method makes it harder to extend your test suite because you have to remember to manually add any newly added test methods to the suite manually. You should take advantage of JUnit's ability to dynamically compute the test suite based on test method names.

**Example**

The following method would be flagged if it appeared within a test case:

```
public static Test suite()
{
...
}
```

**Efficient Expression**

**Severity:** Medium

**Summary**

Some expressions are more efficient than others.

**Description**

This rule finds expressions that can be replaced with other equivalent and more efficient expressions.

**Example**

The expression

```
(new Integer("1234")).intValue()
```

should be replaced by

```
Integer.parseInt("1234")
```

because the latter expression does not create an intermediate Integer object and thus is more efficient.

**Either Nillable Or MinOccurs**

**Severity:** Medium

**Summary**

Never use both nillable ="true" and minOccurs="0" on the same element.

**Description**

This audit rule looks for element declarations that use both nillable ="true" and minOccurs="0".

This results in an ambiguous interface because consumers do not know if the element will exist but be empty or be left out of the instance document all together.

**Example**

The following element declares both nillable ="true" and minOccurs="0" at the same time and would thus be marked as a violation:

```
<xs:element
minOccurs="0"
name="minzeronil"
nillable="true"
type="xs:string"/>
```

**Empty Catch Clause**

**Severity:** Medium

**Summary**

Catch clauses should not be empty.

**Description**

This rule finds places where an exception is caught and nothing is done. It can be configured to allow the existence of a comment to substitute for actual Java code.

**Example**

```
try {
...
} catch (Exception exception) {
}
```

**Empty Class**

**Severity:** Medium

**Summary**

Empty classes should not be declared.

**Description**

This audit rule checks for class declarations that do not include any members (fields, methods, or inner classes). Such classes usually occur if either the implementation was not finished or if the class was being used as a marker. In the latter case the class should be replaced by an interface.

**Example**

The following class definition would be flagged as being a violation:

```
public class EmptyClass
{
}
```

**Empty Do Statement**

**Severity:** Medium

**Summary**

Do statements should not be empty.

**Description**

This rule finds do statements whose body is empty.

**Example**

```
do {
} while(someCondition());
```

**Empty Enhanced For Statement**

**Severity:** Medium

**Summary**

The body of an enhanced for loop should never be empty.

**Description**

This audit rule finds enhanced for loops whose body is empty.

**Example**

```
for (int count : counts) {
}
```

**Empty Finalize Method**

**Severity:** Medium

**Summary**

The body of a finalize method should never be empty.

**Description**

This audit rule finds finalize methods whose body is empty.

**Example**

```
protected void finalize()
{
}
```

**Empty Finally Clause**

**Severity:** Medium

**Summary**

Finally clauses should never be empty.

**Description**

This audit rule finds finally clauses whose block is empty.

**Example**

```
try {
...
} finally {
}
```

**Empty For Statement**

**Severity:** Medium

**Summary**

The body of a for loop should never be empty.

**Description**

This audit rule finds for loops whose body is empty.

**Example**

```
for (int i = 0; i < array.length; i++) {
}
```

**Empty If Statement**

**Severity:** Medium

**Summary**

The clauses of an if statement should never be empty.

**Description**

This audit rule finds if statements whose then or else clauses are empty.

**Example**

```
if (this == that) {
}
```

**Empty Initializer**

**Severity:** Medium

**Summary**

The body of an initializer should never be empty.

**Description**

This audit rule finds initializers whose body is empty.

**Example**

```
static {
}
```

**Empty Method**

**Severity:** Low

**Summary**

Empty methods should never be used.

**Description**

Methods with an empty body usually occur only when someone has forgotten to implement the method. This audit rule finds methods whose body is empty.

**Example**

```
public void doSomething()
{
}
```

**Empty Statement**

**Severity:** High

**Summary**

Empty statements should never be used.

**Description**

This audit rule finds places where an empty statement occurs within a control structure. (An empty statement is a semicolon appearing alone in a place where a statement is allowed). The existence of an empty statement usually indicates a problem, such as a piece of code that was unintentionally removed or a semicolon added in the wrong place.

**Example**

```
if (hasBeenAuthenticated);
grantSpecialAccess();
```

**Empty String Detection**

**Severity:** Medium

**Summary**

The method equals("") should not be used to determine if a String is empty.

**Description**

This audit rule detects instances where Strings are detected to be empty with the <String>.equals("") method. Instances of "".equals(<String>) are also flagged as violations.

**Example**

The following would be flagged:

```
String str = ...;
if(str.equals("")) {
...
}
```

**Empty Switch Statement**

**Severity:** Medium

**Summary**

The body of a switch statement should never be empty.

**Description**

This audit rule finds switch statements whose body is empty.

**Example**

```
switch (value) {
}
```

**Empty Synchronized Statement**

**Severity:** High

**Summary**

Synchronized statements should never be empty.

**Description**

This audit rule finds empty synchronized statements.

**Example**

```
synchronized (monitor) {
}
```

**Empty Try Statement**

**Severity:** Medium

**Summary**

The body of a try statement should never be empty.

**Description**

This audit rule finds try statements whose body is empty.

**Example**

```
try {
```

```
} finally {
file.close();
}
```

**Empty While Statement**

**Severity:** Medium

**Summary**

The body of a while statement should never be empty.

**Description**

This audit rule finds while statements whose body is empty.

**Example**

```
while (index < count) {
}
```

**Enforce Cloneable Usage**

**Severity:** Medium

**Summary**

When an Object is cloned, Object data is copied and returned, because of this secure classes must be conscious of Cloneable.

**Description**

This audit rule aims to prevent adversaries from accessing data by extending classes and creating clone methods. Specifically, this rule flags non-anonymous classes that:
(1) do not implement Cloneable (so that the rule doesn't flag appropriate uses of Cloneable utilities),
(2) are non-final (final classes can't be extended),
(3) do not inherit a clone method (since inserting a clone method would be unnecessarily repetitive),
(4) and do not override clone():

```
public Object clone() throws CloneNotSupportedException;
```

and may enforce the following as the body of the clone method:

```
{
throws java.lang.CloneNotSupportedException("Type not cloneable");
}
```

without, or with, a String input description.

Note: Even though classes that do not implement the Cloneable interface throw CloneNotSupportedException, adversaries are not prevented from extending a class, implementing the Cloneable interface, and then calling clone() to retrieve a copy of an instance of the class.

For more audit rule options concerning clone(), see Clone Method Usage and Override Clone Judiciously under Semantic Errors.

**Security Implications**

Classes that do not override clone() risk allowing a malicious user to subclass and override the clone method allowing them to gain access to data not intended by the application.

**Example**

The classes A, B and C below will not be flagged, but D will be.

```
class A {
public Object clone() {
throw new java.lang.CloneNotSupportedException();
}
}

class B extends A {}

final class C{}

class D {}
```

**Enforce Singleton Property with Private Constructor**

**Severity:** Medium

**Summary**

Flag classes that appear to follow the Singleton pattern but have a non-private constructor.

**Description**

Ensure that a class defined as a singleton follows specific rules that disallow multiple instances to be created. Singleton classes should have a single, private constructor and static access to the instance.

**Entity Bean's Remote Interface**

**Severity:** Medium

**Summary**

An Entity Bean exposed through a remote interface is a performance issue that should be avoided.

**Description**

In most cases, the definition of a remote entity bean is the result of an error in the container configuration and should be avoided.

**Security Implications**

An Entity bean exposed through a remote interface becomes accessible for an attacker that can use the performance issues that are caused by a remote entity bean to perform an attack on the server.

**Example**

The following code would be flagged as a violation because it declares a remote interface for an entity bean:

```
<ejb-jar>
<enterprise-beans>
<entity>
<ejb-name>AccountEntry</ejb-name>
<home>com.somecorp.beans.account.AccountEntryHome</home>
<remote>com.somecorp.beans.account.AccountEntry</remote>
...
</entity>
...
</enterprise-beans>
</ejb-jar>
```

**Entity Beans**

**Severity:** Medium

**Summary**

Entity Beans should be properly defined.

**Description**

This audit rule finds problems with Entity Beans:
- Entity bean should be declared public
- Entity bean name should end with 'Bean'
- Entity bean not implement a finalize() method
- ejbCreate() method should be declared public
- ejbCreate() method should not be declared as static or final
- ejbCreate() method should be declared void
- Entity bean should implement at least one ejbCreate() method
- Entity bean should implement a no-argument constructor
- One ejbPostCreate() method for each ejbCreate() method
- ejbPostCreate() method should be declared public
- ejbPostCreate() method should not be declared as static or final
- ejbPostCreate() method should be declared void
- ejbPostCreate() parameters should match ejbCreate() method
- ejbPostCreate() name should match ejbCreate() name
- ejbFind() method should be declared public
- ejbFind() method should not be declared as static or final
- ejbSelect() method should be declared public
- ejbSelect() method should be declared abstract
- ejbSelect() method should throw the javax.ejb.FinderException exception
- ejbHome() method should be declared public
- ejbHome() method should not be declared static
- ejbHome() method should not throw the javax.ejb.RemoteException exception

**Entry Point Method**

**Severity:** Medium

**Summary**

The main() method should be defined as "public static void main(java.lang.String[])".

**Description**

This audit rule finds main() methods that are not defined as "public static void main(java.lang.String[])". The main() method should only be used as the entry point for a class.

**Example**

The following method would be flagged as a violation because it is not declared to be a static method:

```
public void main(String[] args)
{
...
}
```

**Enumerated Type Naming Convention**

**Severity:** Medium

**Summary**

Enumerated type names should conform to the defined standard.

**Description**

This audit rule checks the names of all enumerated types.

**Example**

If the rule were configured to require that all enumerated type names start with a capital letter, the following declaration would be flagged as a violation:

```
public enum response {stronglyAgree, agree, disagree, stringlyDisagree,
noOpinion}
```

**Enumeration Constant Naming Convention**

**Severity:** Medium

**Summary**

Enumeration constant names should conform to the defined standard.

**Description**

This audit rule checks the names of all enumeration constants.

**Example**

If the rule were configured to require that all enumeration constant names start with a capital letter, the following declaration would be flagged as a violation:

```
public enum response {stronglyAgree, agree, disagree, stringlyDisagree,
noOpinion}
```

**Environment Variable Access**

**Severity:** High

**Summary**

Environment variables should not be accessed because not all platforms have support for environment variables.

**Description**

The method System.getenv() should not be used to access environment variables because not all platforms have support for environment variables. Properties should be used instead.

**Example**

The following invocation of the getenv() method would be flagged as a violation:

```
System.getenv("PATH");
```

**Equality Test with Boolean Literal**

**Severity:** Medium

**Summary**

Boolean literals should never be used in equality tests.

**Description**

This audit rule finds equality tests (using either == or !=) in which either or both of the operands are a Boolean literal (either true or false).

**Example**

```
if (todayIsTuesday() == true) {
...
}
```

**Exception Creation**

**Severity:** Medium

**Summary**

Exceptions should be created with as much information as possible.

**Description**

When exceptions are created they should be given as much information as possible, including a message and, when one is available, the exception that caused the exception being created.

**Example**

The following exception creation would be flagged as a violation because it does not specify a message:

```
throw new CriticalApplicationException();
```

**Exception Declaration**

**Severity:** Medium

**Summary**

Exceptions should be declared to inherit from Exception, but not from either RuntimeException or RemoteException.

**Description**

Exceptions should be declared to inherit from Exception, but not from either RuntimeException or RemoteException.

**Example**

The following class declaration would be flagged as an error because it extends RuntimeException:

```
public class MyException extends RuntimeException
{
}
```

**Exception Parameter Naming Convention**

**Severity:** Medium

**Summary**

Exception parameter names should conform to the defined standard.

**Description**

This audit rule checks the names of all exception parameters.

**Example**

If the rule were configured to only allow names of the form "e", the following parameter would be flagged as a violation:

```
throw {
...
} catch (Exception problem) {
...
}
```

**Explicit "this" Usage**

**Severity:** Medium

**Summary**

Instance fields should, or should not, be accessed using "this".

**Description**

This audit rule checks for the explicit usage of the keyword "this" when accessing instance fields. The rule can be configured to always check for the presence or absence of the keyword.

**Example**

If the rule is configured to disallow using "this" to qualify fields unless necessary, the following expression would be flagged as a violation:

```
public void incrementCount(int amount)
{
this.count += amount;
}
```

**Explicit Invocation of Finalize**

**Severity:** Medium

**Summary**

The method finalize() should never be explicitly called.

**Description**

This audit rule looks for explicit invocations of the method finalize(). The finalize method should only be invoked by the VM.

**Example**

The following method invocation would be flagged as a violation:

```
object.finalize();
```

**Explicit Subclass of Object**

**Severity:** Medium

**Summary**

The class "Object" should not be explicit specified as a superclass.

**Description**

Subclasses of Object should be declared implicitly by omitting the "extends" clause, not explicitly by including "extends Object" in their declaration.

**Example**

The following class declaration would be flagged as a violation:

```
public class Employee extends Object
{
...
}
```

**Expression Evaluation**

**Severity:** Medium

**Summary**

Expression evaluation.

**Description**

This set of audit rules checks the value of expressions for certain conditions. It detects constant and zero values, divide-by-zero, and others.

**Example**

The following expression would be flagged as a violation because it always produces the same value:

```
int secondsPerDay = 24 * 60 * 60;
```

The following expression would be flagged as a violation because it will always cause a divide by zero exception:

```
return 23 / 0;
```

**External Dependency in Ant**

**Severity:** Medium

**Summary**

Using external downloadable dependencies is an additional security risk.

**Description**

This audit rule violates the retrieval of dependancies in an Ant script via<get> ant task.

**Security Implications**

An external repository can be compromised by an attacker to allow malicious code into your application during its building. This threat is completely negated when you do not use downloadable dependencies at all; if this is not an option, you can use local repository proxies for storing trusted dependency packages that can be used during the build. In contrast to external repositories belonding to third parties, such a repository's security can be controlled.

**Example**

The following part of an Ant script would be flagged as a violation because it tries to access a remote dependency jar via <get> task:

```
<target name="get_deps">
<get src="http://ext.repository.com/jackarta/commons-lang.jar"
dest="deps/commons-lang.jar" />
</target>
```

**External Dependency in Ivy**

**Severity:** Medium

**Summary**

Using external repositories is an additional security risk.

**Description**

This audit rule violates the usage of external Ivy repositories, declared in the chain of resolvers, other than from the list of permitted ones.

**Security Implications**

An external repository can be compromised by an attacker to allow malicious code into your application during its building. This threat is completely negated when you do not use downloadable dependency managers at all; if this is not an option, you can use local repository proxies for storing trusted dependency packages that can be used during the build. In contrast to external repositories belonging to third parties, such a repository's security can be controlled.

**Example**

The following Ivy `ivysettings.xml` file declares the only one `default` resolver. If `http://ivy.internal.repo/` URL is present in the list of secure repositories, these settings will pass the test. Otherwise, this file will be marked as a violation:

```
<ivysettings>
<settings defaultResolver="default"/>
<properties file="${ivy.settings.dir}/ivysettings.properties" />
<resolvers>
<chain name="default">
<url name="internal" checkmodified="true">
<ivy pattern="http://ivy.internal.repo/[org]/[mod]/ivy-[rev].xml"/>
<artifact pattern="http://ivy.internal.repo/[org]/[mod]/[art]-[rev].[type]"/>
</url>
</chain>
</resolvers>
</ivysettings>
```

**External Dependency in Maven**

**Severity:** Medium

**Summary**

Using external repositories is an additional security risk.

**Description**

This audit rule violates the usage of external Maven repositories other than from the list of permitted ones.

Remember that tools such as Maven store a list of repositories plugged in by default. You should redeclare these too.

**Security Implications**

An external repository can be compromised by an attacker to allow malicious code into your application during its building. This threat is completely negated when you do not use downloadable dependency managers at all; if this is not an option, you can use local repository proxies for storing trusted dependency packages that can be used during the build. In contrast to external repositories belonding to third parties, such a repository's security can be controlled.

**Example**

The following part of a Maven script would be flagged as a violation because it accesses a dependency from the repository:

```
<repositories>
<repository>
<id>apache.incubator</id>
<url>http://people.apache.org/repo/m2-incubating-repository</url>
</repository>
</repositories>
```

**Extra Semicolon**

**Severity:** Low

**Summary**

Extra semicolons clutter the code and serve no useful purpose.

**Description**

This audit rule finds places where a semicolon occurs but is not needed. While not strictly an error, such semicolons clutter the code and serve no useful purpose.

**Example**

```
while (index < count) {
index = index + 1;;
};
```

**Fail Invoked in Catch**

**Severity:** Medium

**Summary**

Exceptions representing failure should not be caught.

**Description**

This audit rule checks for test methods that catch exceptions in order to cause the test to fail by invoking one of the fail() methods. It is not necessary to catch the exception because uncaught exceptions will automatically cause the test to fail.

**Example**

The following invocation of the fail() method would be flagged as a violation because it occurs within a catch block:

```
try {
...
} catch (Exception exception) {
fail("exception thrown");
}
```

**Favor Static Member Classes over Non-Static**

**Severity:** Medium

**Summary**

Member classes should be defined as static classes when possible.

**Description**

This rule identifies member classes that are not defined as static classes but that could possibly be defined as such.

**Field Access Protection**

**Severity:** Medium

**Summary**

When a field that is usually accessed in a synchronized context is accessed without synchronization, this indicates an error.

**Description**

This audit rule looks for unsynchronized accesses to fields that are usually accessed in a synchronized way. A field is considered to match the criteria if synchronization is used to access it in at least 66% of all of the places in which it is accessed.

**Security Implications**

Partial or incomplete synchronization is the primary source of all deadlocks and race conditions. Multi-threaded parts of data should always be accessed in a synchronized block.

**Example**

The following class appears to provide synchronized access to the internal list, but `remove()` method is not synchronized and thus will be marked as violation:

```
public class SynchronizedList {
private final List data = new ArrayList();
public synchronized void add(Object item) {
data.add(item);
}
public synchronized int size() {
return data.size();
}
public synchronized Object pop() {
return data.remove(0);
}
public void remove(Object item) {
data.remove(item);
}
}
```

**Field Javadoc Conventions**

**Severity:** Low

**Summary**

All fields should have a Javadoc comment associated with them.

**Description**

This audit rule checks for the existence of a Javadoc comment for each field. It can be configured to only flag fields with the specified visibilities.

**Example**

The following field declaration would be flagged as a violation because it does not have a Javadoc comment associated with it:

```
private String name;
```

**Field Might Have Null Value**

**Severity:** Medium

**Summary**

You should check fields used in methods beacuse they might have `null` value.

**Description**

This audit rule looks for references to fields whose value can be `null` where the value of the field is not checked before being dereferenced.

**Security Implications**

Use checks on a null pointer because `NullPointerException` might be thrown.

**Example**

The following usage of the field `date` will be marked as a violation beacuse it is not checked:

```
public class TestClass {
private Date date = null;
public void badUsage() {
String myStr = date.toString();
}
}
```

**Field Only Used in Inner Class**

**Severity:** Medium

**Summary**

Instance fields that are only used in an inner class should be defined in the inner class.

**Description**

This audit rule looks for private instance fields that are only being used within a single inner class. There are valid reasons for structuring code this way, such as when the lifetime of the field needs to be longer than the lifetime of instances of the inner class or when the field needs to be shared by some (but not all) of the instances of the inner class. Generally, however, such fields should be declared by the inner class.

**File Comment**

**Severity:** Low

**Summary**

Compilation units should have a file comment.

**Description**

This audit rule finds compilation units that do not have a comment as the first element in the file.

**Example**

A file that begins with a package statement would be flagged as a violation because a comment is expected to appear before anything else.

**File Length**

**Severity:** Low

**Summary**

Compilation units should not be too long.

**Description**

This audit rule finds compilation units that are longer than a specified number of lines.

**Example**

If the rule is configured to allow files of up to 40,000 lines, a file containing 100,000 lines of code would be flagged as a violation.

**Filename Given Out**

**Severity:** Medium

**Summary**

The name of a locally used file is valuable information for an attacker.

**Description**

This rule violates printing file names and paths to the HttpServletResponse output stream.

**Security Implications**

If the name of a locally used file is displayed to the user over a web-interface, this information could be used by an attacker to retrieve valuable information from the system. Such an action should therefore be avoided.

**Example**

The following code would be flagged as a violation because it displays a filename to the user:

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
File file = File.createTempFile("somefile", ".tmp");
if (debug.equals("true")) resp.getOutputStream().print("Using " +
file.toString());
...
```

**Final Method Parameter In Interface**

**Severity:** Medium

**Summary**

Method parameters in interfaces should not be final.

**Description**

This audit rule flags any final method parameters defined in interfaces. It is unnecessary to mark an interface methods's parameters as final, since there is no implementation associated with an interface, and making a parameter final only affects the implementation.

**Example**

The following declaration of the final parameters a and b would be flagged:

```
interface foo {
public void add(final int a, final int b);
}
```

**Finalize Method Definition**

**Severity:** Medium

**Summary**

Finalize methods should not have parameters or a non-void return type.

**Description**

The only way to declare a finalize method is

```
protected void finalize() [throws Throwable]
```

You can create other finalize methods that take parameters, but they will not be called automatically by the system, and may confuse anyone reading the code. You should reserve the name finalize for the real finalize method. This audit rule finds finalize() methods that have parameters or do not have a void return type.

**Example**

The following method declaration would be flagged as a violation because the method returns an integer:

```
protected int finalize()
{
...
}
```

**Finalize Should Not Be Public**

**Severity:** Medium

**Summary**

Finalize methods declared within an Applet should not be public

**Description**

This audit rule flags any declarations of the Object.finalize that are public and within an Applet. If the finalize method is declared properly, then the method should not need to be public.

**Security Implications**

Malicious users can perform attacks on Applets by calling public finalize methods.

**Example**

The following declaration of finalize would be flagged:

```
class A extends java.applet.Applet {
public void finalize() {}
}
```

**Float Comparison**

**Severity:** Medium

**Summary**

Floating-point values should not be compared using equals (==) or not equals (!=).

**Description**

This audit rule finds places where two floating-point values are compared using either the equals (==) or not equals (!=) operators. The problem is that floating-point values are not exact, and floating-point operations sometimes introduce rounding errors. This sometimes results in getting the wrong result from equality-based comparisons.

**Example**

Given two floating point variables:

```
double oneThird = 1.0 / 3.0;
double anotherThird = (2.0 / 3.0) - oneThird;
```

The following expression would be flagged as a violation:

```
if (oneThird == anotherThird)
```

**Floating Point Use**

**Severity:** Medium

**Summary**

Floating point values should not be used.

**Description**

This audit rule checks for uses of floating point values. It finds such uses as the declared type of variables, the return type of methods, literal values, references to floating point valued variables, and the invocation of methods that return floating point values. Floating point values should rarely be used because of the potential for rounding errors.

**Example**

The following declaration would be flagged as a violation:

```
private float accountBalance;
```

**Form Does Not Extend Validator Class**

**Severity:** High

**Summary**

Struts forms that are defined to use validation should extend a validator class.

**Description**

In order to use the Struts Validator, a form must extend one of the following classes:

```
org.apache.struts.validator.DynaValidatorActionForm
org.apache.struts.validator.DynaValidatorForm
org.apache.struts.validator.ValidatorActionForm
org.apache.struts.validator.ValidatorForm
```

You must extend one of these classes because the Struts Validator ties in to your application by implementing the validate() method in these classes.Forms derived from the following classes cannot use the Struts Validator:

```
org.apache.struts.action.ActionForm
org.apache.struts.action.DynaActionForm
```

**Example**

If the struts-config.xml file contained the following tag and the class example.NotValidatorForm does not extends one of the validator classes listed above, the bean notValidatorForm would be flagged:

```
<form-bean name="notValidatorForm" type="example.NotValidatorForm"/>
```

**Fully Parenthesize Expressions**

**Severity:** Medium

**Summary**

All nested expressions should be fully parenthesized.

**Description**

This audit rule checks for nested binary expressions that are not fully parenthesized and flags them.

**Example**

The two multiplication expressions in the statement below should each be parenthesized to make clear the order of precedence:

```
result = x * x + y * y;
```

**Getter and Setter Methods Should Be Final**

**Severity:** Medium

**Summary**

Getter and setter methods should be declared final.

**Description**

This rule looks for getter and setter methods that are not declared final.

**Security Implications**

An attacker can modify the behavior of the class by overriding the getter or the setter. This can lead to unexpected and insecure behavior.

**Example**

The following code would be flagged as a violation because its getter method is not declared final:

```
public class Person {
private String name;
public String getName() {
return name;
}
}
```

**Handle Numeric Parsing Errors**

**Severity:** Medium

**Summary**

Numeric parsing errors should be handled where they occur.

**Description**

This audit rule finds invocations of methods that parse numeric values from Strings (and hence can throw a NumberFormatException) where the exception is not handled (caught) in the same scope.

**Example**

The following invocation of parseInt would be flagged because it is not wrapped in a try statement that catches NumberFormatException:

```
int value = Integer.parseInt("42");
```

**Hardcoded Password**

**Severity:** Medium

**Summary**

Passwords should not be hardcoded.

**Description**

This audit rule violates any instance where a hardcoded password is passed to `java.util.Properties` or `java.sql.DriverManager`. If a such a password is found, and it is also empty, a violation is thrown to create a password.

**Security Implications**

When a password is hardcoded in an application, not only can other developers see the password, but the password can also be extracted from the java byte code.

**Example**

The line "properties.setProperty("password", "somePassword");" would be flagged as "somePassword" is a hardcoded password.

```
Properties properties = new Properties();
properties.setProperty("user", "someUserName");
properties.setProperty("password", "somePassword");
DriverManager.getConnection(..., properties);
```

**Hiding Inherited Fields**

**Severity:** Medium

**Summary**

Inherited fields should not be hidden.

**Description**

This audit rule finds fields that hide inherited fields. That is, it finds fields that have the same name as a visible field from a superclass. Note that private fields are not visible in subclasses, so declarations of fields with the same name as a private field declared in a superclass are not flagged.

**Example**

Given a class declaration like the following:

```
public class Person
{
protected String name;
...
}
```

The field declaration in the following class declaration would be flagged as a violation:

```
public class Employee extends Person
{
protected String name;
...
}
```

**Hiding Inherited Static Methods**

**Severity:** Medium

**Summary**

Inherited static methods should not be hidden.

**Description**

This audit rule finds methods that hide inherited static methods. That is, it finds methods that have the same name and compatible argument types as a static method from a superclass. The problem with defining methods like this is that it is too easy to miss the fact that the methods are static and that the subclasses method therefore does not override the method from the superclass.

**Example**

Given a class declaration like the following:

```
public class Person
{
public static Person newNamed(String name)
{
...
}
...
}
```

The method declaration in the following class declaration would be flagged as a violation:

```
public class Employee extends Person
{
public static Person newNamed(String name)
{
...
}
...
}
```

**HTTP Response Splitting**

**Severity:** High

**Summary**

User input might be getting input directly into a HTTP response.

**Description**

HTTP Response Splitting occurs when the user is able to enter data directly into an HTTP response.

To detect violations, this audit rule searches the code for statements such as `HTTPServletResponse.sendRedirect(..)` and traces where the redirection data could have come from. In cases where the source of the redirect location is user input, such as data from a servlet request, `javax.servlet.ServletRequest.getParameter(java.lang.String)`, a violation is created.

These two sets of methods, the locations where tainted user data can come from and the HTTP response methods, are editable by the user. Note, the Cookie constructor is included since HTTP responses can include Cookie information through the `HttpServletResponse.addCookie(..)` method. If methods are missing that are in a common package (such as `java.lang.*`), please let CodePro support know.

Finally, cleaning methods such as `HttpServletResponse.encodeRedirectURL(String)`, are taken into account and thus if user input is cleaned, it won't get flagged when passed to a HTTP response.

**Security Implications**

By having direct access to enter data directly into an HTTP response, a malicious user could potentially split the response by including a carriage return or a line feed, which can lead to a number of vulnerabilities.

**Example**

The invocation of the method `sendRedirect(..)` would be flagged as a violation since it puts the first name information passed from a servlet request directly into a HTTP response:

```
ServletRequest servletRequest = ...;
HttpServletResponse httpServletResponse = ...;

String firstName = servletRequest.getParameter("firstName");
httpServletResponse.sendRedirect(firstName);
```

**Illegal Main Method**

**Severity:** Medium

**Summary**

A main method should only be declared in application classes.

**Description**

This audit rule checks for declarations of a main method occurring in non-application classes. This is most commonly done in order to write testing code, but testing code should be written in a test case.

**Implement a Zero-Argument Constructor**

**Severity:** High

**Summary**

Implement a zero-argument constructor for persistent classes.

**Description**

This audit rule looks for persistent classes that do not explicitly implement a zero-argument constructor. The constructor does not need to be public, but it needs to exist so that Hibernate can instantiate the class using Constructor.newInstance().

**Example**

The following class would be flagged because it has no explicitly defined constructor.

```
public class Message {
private Long id;
private String text;
private Message nextMessage;

public Message(String text) {
this.text = text;
}

public Long getId() {
return id;
}
private void setId(Long id) {
this.id = id;
}
}
```

**Implement BeanNameAware Interface**

**Severity:** Low

**Summary**

Implement the BeanNameAware interface in your bean classes.

**Description**

This audit rule looks for beans that do not implement the BeanNameAware interface. Being able to have a bean find out its name at runtime is really useful for logging. Consider a situation where you have many beans of the same type running under different configurations. The bean name can be included in log messages to help you differentiate between which one is generating errors and which ones are working fine when something goes wrong.

Implementation is fairly trivial and no special configuration is required to take advantage of the BeanNameAware interface.

**Example**

Implementation looks like following:

```
public class LoggingBean implements BeanNameAware {
private String beanName = null;

public void setBeanName(String beanName) {
this.beanName = beanName;
}
}
```

**Implement Iterable**

**Severity:** Low

**Summary**

Classes which define an `iterator()` method should implement Iterable.

**Description**

This audit rule checks for classes which define an `iterator()` method,but do not implement the `Iterable` interface. Implementing this interface allows the class to be used in an enhanced for loop. In these cases, it takes no extra effort to implement `Iterable`.

**Example**

The following class declaration would be flagged as a violation:

```
public class Foo
{
public Iterator iterator() {...}
}
```

**Implicit Subclass of Object**

**Severity:** Medium

**Summary**

The class "Object" should be explicit specified as a superclass.

**Description**

Subclasses of Object should be declared explicitly by including "extends Object" in their declaration, not implicitly by omitting the "extends" clause.

**Example**

The following class declaration would be flagged as a violation:

```
public class Employee
{
...
}
```

**Import of Implicit Package**

**Severity:** Medium

**Summary**

Packages that are implicitly imported should never be explicitly imported.

**Description**

This audit rule checks to ensure that none of the import declarations imports from a package that is implicitly imported (either java.lang or the package containing the compilation unit being audited).

**Example**

```
import java.lang.*;
```

**Import Order**

**Severity:** Medium

**Summary**

Import declarations should be consistently ordered.

**Description**

This audit rule finds import declarations that are not in the specified order.

**Example**

If the order of the import declarations has been configured such that packages that start with "com" should appear before packages that begin with "org", the following imports would be flagged as being in the wrong order:

```
import org.eclipse.core.resources.*;
import com.instantiations.codePro.*;
```

**Import Style**

**Severity:** Medium

**Summary**

Only one style of import should be used.

**Description**

This audit rule checks to ensure that all of the import declarations have the same style, either demand imports (such as "java.util.*"), or explicit imports (such as "java.util.List").

**Example**

If the rule is configured such that only explicit imports are allowed, the following import would be flagged as a violation:

```
import java.util.*;
```

**Improper calculation of array hashCode**

**Severity:** Medium

**Summary**

Because the `hashCode()` method of an array returns the identityhashCode, this method should not be used to generate hash codes for arrays.

**Description**

This audit rule looks for invocations of `hashCode()` on arrays, as well as common functions that will invoke `hashCode()` on an array.

**Example**

The following would be flagged as a violation:

```
new Person[] {
new Person("Alice"),
new Person("Bob"),
new Person("Charlie")
}.hashCode();
```

**Improper conversion of Array to String**

**Severity:** Medium

**Summary**

Because the `toString()` method of an array does not generate useful information, some functions that accept object parameters will display useless results when an array is passed to them.

**Description**

This audit rule looks for invocations of `toString()` on arrays, as well as common functions that will convert an array to a String using the `toString()` method. `toString()` does not return the contents of the array in a useful format, instead, it generates a string similar to `[C@16f0472`.

**Example**

The following would be flagged as a violation:

```
System.out.println(new Person[] {
new Person("Alice"),
new Person("Bob"),
new Person("Charlie")
});
```

**Improper Use of Thread.interrupted()**

**Severity:** Low

**Summary**

The static method `Thread.interrupted()` can be easily confused with the instance method `isInterrupted()`.

**Description**

Since `interrupted()` is a static method which returns whether the *current* thread is interrupted, it should always be invoked using `Thread.interrupted()`. This audit rule looks for any invocations of `interrupted()` on instances of `Thread`.instead.

**Example**

The following would be flagged as a violation:

```
new Thread().interrupted()
```

**Inappropriate Language**

**Severity:** Medium

**Summary**

Inappropriate language should not be used in the source code.

**Description**

This audit rule finds uses of inappropriate language within the source code.

**Include Implementation Version**

**Severity:** Medium

**Summary**

Manifest files should include an implementation version.

**Description**

This audit rule checks for manifest files that do not include an implementation version, or include an implementation version with the wrong format. An implementation version is specified by defining an attribute with the name "Implementation-Version". The value of the attribute should consist of three integers, separated by periods, representing a major, minor, and micro version, in that order.

**Example**

The following implementation version would be flagged as a violation because the version number is not well formed:

```
Implementation-Version: 2.3a
```

**Incompatible Renderer Type**

**Severity:** Medium

**Summary**

The method getRendererType must return a renderer compatible with the type of component specified by the getComponentType method.

**Description**

This audit rule looks for subclasses of UIComponentTag that return a literal string or a string constant from getComponentRenderer() such that the returned string denotes a renderer which cannot render the component specified by the getComponentType() method's return value.

**Example**

Given the following XML:

```
<render-kit>
<renderer>
...
<component-family>FamilyA</component-family>
<renderer-type>RendererA</renderer-type>
<renderer-class>renderers.RendererA</renderer-class>
</renderer>
</render-kit>
<component>
<component-type>ComponentB</component-type>
<component-class>components.ComponentB</component-class>
<component-extension>
```

```
<component-family>FamilyB</component-family>
</component-extension>
</component>
```

and the following component code:

```
public class ComponentB extends UIOutput {
...
public String getFamily() {
return "FamilyB";
}
...
}
```

the return value of the getRendererType() method of the following class will be flagged as violation because the specified renderer cannot render components of the specified family.

```
public class ViolatingTag extends UIComponentTagBase {
...
public String getRendererType() {
return "RendererA";
}
public String getComponentType() {
return "ComponentB";
}
...
}
```

**Incompatible types stored in a collection**

**Severity:** Medium

**Summary**

You should avoid incompatible casts because a `ClassCastException` will be thrown.

**Description**

This audit rule looks for places in the code that cast elements retrieved from a collection to a type that is not compatible with the type of elements being put into that very collection.

**Security Implications**

Incompatible cast will cause a `ClassCastException` to be thrown. This could be used to create a potential denial-of-service state or reveal security-sensitive parts of an application's design through the stack trace.

**Example**

The following invocation of the `get()` method will be marked as a violation because its return value is cast to a type incompatible with the one being put into collection:

```
public class MyClass {
...
private List testList;
...
public void myMethod(MyClass obj) {
testList.add(obj);
Integer test = (Integer)testList.get(0);
}
```

**Incomplete reset()**

**Severity:** Medium

**Summary**

Reset each member variable in the reset() method.

**Description**

If the reset() method is implemented, this audit rule checks to make sure that each member variable is reset.

**Example**

The reset method below would be flagged because the username field is not reset.

```
public class LoginForm extends ActionForm {
private String password = null;
private String username = null;

...

public void reset(ActionMapping mapping,
HttpServletRequest request) {
this.password = null;
}
}
```

**Incomplete State Storing**

**Severity:** Medium

**Summary**

The methods `saveState()` and `restoreState()` should both be implemented at the same level of class hierarchy.

**Description**

It is improper to implement the `saveState()` and `restoreState()` methods of a `StateHolder` on different levels of a class hierarchy. This approach is error-prone and should be avoided. This rule looks for classes that implement only one or the other of these method but not both.

**Security Implications**

This approach is error-prone and thus can aid an attacker.

**Example**

The following class implements only one of the two state methods and will be marked as violation:

```
public abstract class AbstractStateHolder implements StateHolder {
public Object saveState(FacesContext ctx) {
...
}
}
```

**Incomplete Validation Method**

**Severity:** Medium

**Summary**

Validation methods should either invoke setValid() or throw ValidatorException.

**Description**

This audit rule looks for validation methods that neither invoke setValid() nor throw ValidatorException, where a validation method is either the processValidation method defined by the Validator interface or a method with a signature of (FacesContext, UIComponent, Object) in a backing bean. These are the two ways for a validation method to signal invalid input to the framework (and if the input cannot be invalid, the validator is not needed).

**Example**

The following validator will be flagged as a violation because it neither calls UIInput.setValid() nor throws a ValidatorException.

```
public class MyValidator implements Validator {
public void validate(FacesContext context, UIComponent comp, Object value) {
System.out.println("Validating value " + value);
}
}
```

**Inconsistent Conversion Using toArray()**

**Severity:** Medium

**Summary**

Inconsistent conversion using `toArray()` will cause a `ClassCastException`.

**Description**

This audit rule finds places where `toArray()` is invoked, but the result is cast to an incompatible type.

**Example**

The following would be flagged as a violation:

```
List foo = new ArrayList();
(String[]) foo.toArray(new Integer[0]);
```

**Inconsistent Use of Override**

**Severity:** Medium

**Summary**

The Override annotation should be used for all overridden methods.

**Description**

This audit rule finds classes that use the Override annotation for some overridden methods but not for others and flags those for which it is missing.

**Inconsistent Validator Attribute**

**Severity:** Medium

**Summary**

A validator's attribute type must match the configuration file.

**Description**

This audit rule looks for accessor methods (getters and setters) in implementations of the Validator interface whose return type or parameter type (as appropriate) does not match the type specified in the configuration file.

**Example**

Given the following fragment in the faces-config.xml file:

```
<validator>
<description>Checks if the given value matches the ultimate
answer.</description>
<validator-id>MyValidator</validator-id>
<validator-class>validators.MyValidator</validator-class>
<attribute>
<description>...</description>
<attribute-name>pattern</attribute-name>
<attribute-class>java.lang.String</attribute-class>
</attribute>
</validator>
```

The method getPattern() of the following validator will be flagged as a violation because it's return type does not match the attribute type specified in the configuration.

```
public class MyValidator implements Validator {
public Integer getPattern() {
return new Integer(42);
}
...
}
```

**Incorrect Argument Type**

**Severity:** Medium

**Summary**

The actual type of an argument is incorrect.

**Description**

Several methods defined in the interface java.util.Map (and java.util.concurrent.ConcurrentMap) are declared with parameters of type Object even though they are expected to be of the same type as either the keys or the values. This audit rule looks for invocations of these method in which the argument type does not conform with the expected type. Even though the compiler can't identify such invocations as an error, they are almost always wrong.

Specifically, the rule looks for invocations of the following methods:

```
java.util.Map.containsKey(Object)
java.util.Map.containsValue(Object)
java.util.Map.get(Object)
java.util.Map.remove(Object)
java.util.concurrent.ConcurrentMap.remove(Object, Object)
```

**Example**

Given the following declaration:

```
Map<String, String> nameMap;
```

The following invocation would be flagged as a violation:

```
nameMap.get(new Integer(42))
```

**Incorrect Use of equals() and compareTo()**

**Severity:** Medium

**Summary**

If `compareTo()` is overridden for a type, then `equals()` should be overridden as well.

**Description**

When implemented, the `compareTo()` method should override `compareTo(Object)`, and it should be consistent with `equals()`. That is, `a.compareTo(b)` should return `0` if and only if `a.equals(b)` returns `true`.

Specifically, this rule flags cases where `compareTo()` is overridden but `equals()` is not and cases where `compareTo()` is overloaded instead of overridden.

**Example**

The following would be flagged as a violation, since it overrides compareTo, but not equals:

```
public class MyClass {
public int compareTo(Object o) {
return true;
}
}
```

**Indent Code Within Blocks**

**Severity:** Low

**Summary**

Code within blocks should be indented one level more than the block.

**Description**

This audit rule checks for blocks of code that are not indented one level more than the code containing the block.

**Example**

The statements inside the following if statement would be flagged as needing to be indented:

```
if (employee.isHourly()) {
computeHourlyBonus(employee);
} else {
computeSalariedBonus(employee);
}
```

**Index Arrays with Ints**

**Severity:** Medium

**Summary**

Arrays should be indexed with int values.

**Description**

Arrays should be indexed with int values in order to avoid the run-time overhead of converting a shorter type.

**Example**

The array index expression in the following code would be flagged as a violation because the loop variable "b" should be declared to be an int:

```
for (byte b = 0; b < 128; b++) {
array[b] = null;
}
```

**Inefficient use of toArray()**

**Severity:** Low

**Summary**

Passing a zero-length array to `toArray()` is inefficient.

**Description**

When converting a `Collection` to an array using `toArray()`, it is most efficient to pass in an array whose length is equal to the `Collection`'s size. This rule flags locations an array of some fixed length is passed in.

**Example**

The use of toArray in the following code would be flagged:

```
ArrayList foo = new ArrayList();
foo.toArray(new String[0]);
```

**Initialize Static Fields**

**Severity:** Medium

**Summary**

All static fields should be initialized.

**Description**

This audit rule looks for static fields that are not initialized. A static field can be initialized either as part of its declaration or in a static initializer.

**Example**

Assuming that there are no static initializers in the class containing the following static field declaration, it would be flagged because the field is not initialized:

```
private static HashMap instanceMap;
```

**Instance Field Naming Convention**

**Severity:** Medium

**Summary**

Instance field names should conform to the defined standard.

**Description**

This audit rule checks the names of all instance fields.

**Example**

If the rule were configured to only allow instance fields to begin with a lower case letter, the following declaration would be flagged as a violation because it begins with an upper case letter:

```
private int MaxCount;
```

**Instance Field Security**

**Severity:** Medium

**Summary**

Refrain from using non-final public instance fields.

**Description**

To the extent possible, refrain from using non-final public instance fields. Instead, let the interface to your instance field be through accessor methods. In this way it is possible to add centralized security checks, if required.

**Security Implications**

Public, non-final fields are accessible and changeable from anywhere within the application making them potential targets of malicious users.

**Example**

The following field declaration would be flagged as a violation because it is both public and non-final:

```
public int width;
```

**Instance Field Visibility**

**Severity:** Medium

**Summary**

Instance fields should have an appropriate visibility.

**Description**

This audit rule checks the visibility of all non-static fields to ensure that it is one of the allowed visibilities.

**Example**

If the rule were configured to only allow private instance fields, then the following field declaration would be flagged as a violation because it is declared as being public:

```
public int x;
```

**Integer Division in a Floating-point Expression**

**Severity:** Medium

**Summary**

Integers should be converted to floats before division if the result will be converted.

**Description**

When integer values are divided, any remainder is truncated. If the result of that division is going to be converted to a floating-point value, one of the integers should probably be cast to that same floating-point type in order to avoid the rounding error.

**Example**

The following division would be flagged as a violation:

```
int a, b;
float result;
result = a / b;
```

**Interface Naming Convention**

**Severity:** Medium

**Summary**

Interface names should conform to the defined standard.

**Description**

This audit rule checks the names of all interfaces.

**Example**

If the rule were configured to require that all interface names start with a capital "I" and another capital letter, the following declaration would be flagged as a violation because the name does not begin with a capital "I":

```
public interface EventListener
{
...
}
```

**Invalid Check For Binding Equality**

**Severity:** Medium

**Summary**

Bindings should be checked for equality using their keys.

**Description**

This audit rule looks for places where bindings are compared using either the identity operator (==) or the equals method. Bindings are not unique across AST structures, so they should always be compared using their keys.

**Example**

The following comparison would be flagged:

```
if (leftBinding == rightBinding) {
```

**Invalid Check For Java Model Identity**

**Severity:** Medium

**Summary**

Java model elements should be checked for equality using equals.

**Description**

This audit rule looks for places where Java model elements (subclasses of `IJavaElement`) are compared using one of the identity operators (== or !=). Java model elements are not necessarily unique, but equals is guaranteed to be correct.

**Example**

The following comparison would be flagged:

```
IMethod leftMethod, rightMethod;

if (leftMethod == rightMethod) {
```

**Invalid DBC Tag Value**

**Severity:** Medium

**Summary**

The value of a DBC tag must have valid syntax.

**Description**

This audit rule checks the value of all DBC tags (@pre, @post, @inv, and @invariant) to ensure that they have a valid syntax. The value must be either a valid Java expression or two valid Java expressions enclosed in parentheses and separated by a comma.

**Example**

The following precondition would be flagged as a violation because its value is not a valid Java expression:

```
@pre 0 <= value <= 255
```

**Invalid Form Bean Class**

**Severity:** High

**Summary**

Classes used as form beans must subclass ActionForm.

**Description**

All classes declared to be a form bean using the form-bean tag should be real form classes (be subclasses of ActionForm).

**Example**

If the class com.instantiations.struts.example.actionforms.LoginForm did not extended the class ActionFrom, the following line would be flagged:

```
<form-bean
name="loginForm"
type="com.instantiations.struts.example.actionforms.LoginForm"/>
```

**Notes**

You can also use one of the following Struts forms:

```
org.apache.struts.validator.ValidatorForm
org.apache.struts.mock.MockFormBean
org.apache.struts.action.DynaActionForm
org.apache.struts.validator.DynaValidatorForm
org.apache.struts.validator.DynaValidatorActionForm
org.apache.struts.validator.BeanValidatorForm
org.apache.struts.validator.LazyValidatorForm
org.apache.struts.validator.ValidatorActionForm
```

**Invalid Loop Construction**

**Severity:** Medium

**Summary**

Loops should be properly bounded.

**Description**

This audit rule checks for loops whose initial and/or final values could allow the index to go outside the bounds of the collection being accessed within the body of the loop.

**Example**

The following loop would allow the loop variable to take on a value of `array.length`, causing an IndexOutOfBoundsException to be thrown:

```
for (int i = 0; i <= array.length; i++) {
System.out.println(" [" + i + "] = " + array[i]);
}
```

**Invalid Property Type Mapping**

**Severity:** Medium

**Summary**

Map a Hibernate property type only to the corresponding Java type.

**Description**

This audit rule looks for properties whose type is a standard type and checks to make sure that the actual type of the corresponding field is appropriate for the property's type. Non-standard types are ignored.

The following is a table of standard property types and the appropriate types to which they can be mapped.

Property Type Java Type
integer int, java.lang.Integer
long long, java.lang.Long
short short, java.lang.Short
float float, java.lang.Float
double double, java.lang.Double
character char, java.lang.Character
byte byte, java.lang.Byte
boolean, yes_no, true_false boolean, java.lang.Boolean
string, text java.lang.String
date, time, timestamp java.util.Date
calendar, calendar_date java.util.Calendar
big_decimal java.math.BigDecimal
locale java.util.Locale
timezone java.util.TimeZone
currency java.util.Currency
class java.lang.Class
binary byte arrays
clob java.sql.Clob
blob java.sql.Blob

**Example**

Given a class Message containing the following field declaration

```
private String text;
```

The folowing property in the file Message.hbm.xml would be flagged, because the property type "byte" doesn't correspond with Java type "String".

```
<property name="text" column="MESSAGE_TEXT" type="byte"/>
```

**Invalid Source for Database Connection**

**Severity:** Medium

**Summary**

The methods `DriverManager#getConnection()` and `Driver#connect()` are the low-level ways of getting a connection to a database. `DataSource` should be used instead.

**Description**

`DataSource` implementations provided by the majority of application servers provide advanced features such as connection pooling. These features can be accessed automatically when using the class `DataSource`.

**Security Implications**

Using direct connection retrieval can potentially create performance issues that can be used by an attacker to perform a Denial of Service attack.

**Example**

The following code would be flagged as a violation because it directly retrieves the connection:

```
public Connection getConnection() {
return DriverManager.getConnection(dbUrl);
}
```

**Invalid Visitor Usage**

**Severity:** Medium

**Summary**

A visit and endVisit methods should only be invoked by the object being visited.

**Description**

This audit rule looks for places where a `visit` or `endVisit` method is being invoked by an object other than the one being visited. Such invocations indicate that the visitor pattern is being

used incorrectly. Clients should not invoke the `visit` and `endVisit` methods directly, but should ask the object to be visited to accept the visitor.

**Example**

The following invocation of the `visit` method would be flagged as a violation because the argument to the invocation is not `this`:

```
visitor.visit(someObject);
```

**Invocation of Default Constructor**

**Severity:** High

**Summary**

Default constructors should not be invoked within subclass constructors.

**Description**

This audit rule looks for invocations of the default constructor for a superclass within a constructor in the subclass. The invocation is unnecessary because the compiler will automatically add such a call.

**Example**

The following constructor invocation would be flagged as a violation:

```
public Point(int x, int y) {
super();
...;
}
```

**Invoke super.finalize() from within finalize()**

**Severity:** Medium

**Summary**

Every implementation of finalize() should invoke super.finalize().

**Description**

This audit rule looks for implementations of the method finalize() that do not invoke the inherited finalize() method.

**Example**

The following definition of the method finalize() would be flagged because it does not invoke the inherited implementation of finalize():

```
protected void finalize()
throws Throwable
{
if (fileReader != null) {
fileReader.close();
fileReader = null;
}
}
```

**Invoke super.release() Within release()**

**Severity:** Medium

**Summary**

An overriden release() method should invoke the superclass implementation.

**Description**

This audit rule looks for release() methods implemented by custom tag handlers that do not invoke super.release() in the body of the method.

**Example**

The release() method of the following class will be flagged as a violation because it does not contain an invocation of super.release().

```
public class MyTag extends UIComponentClassicTagBase {
private String foo;
public void release() {
foo = null;
}
}
```

**Invoke super.setUp() from within setUp()**

**Severity:** Medium

**Summary**

Every implementation of setUp() should invoke super.setUp().

**Description**

This audit rule looks for implementations of the method setUp() that do not invoke the inherited setUp() method.

**Example**

The following definition of the method setUp() would be flagged because it does not invoke the inherited implementation of setUp():

```
public void setUp()
{
employee = new Employee("Jane Doe");
}
```

**Invoke super.tearDown() from within tearDown()**

**Severity:** Medium

**Summary**

Every implementation of tearDown() should invoke super.tearDown().

**Description**

This audit rule looks for implementations of the method tearDown() that do not invoke the inherited tearDown() method.

**Example**

The following definition of the method tearDown() would be flagged because it does not invoke the inherited implementation of tearDown():

```
public void tearDown()
{
```

```
employee = null;
}
```

**Invoke super.validate() in validate()**

**Severity:** Medium

**Summary**

Always call super.validate() in your validate methods in ActionForm and check is super.validate() return null or not.

**Description**

The validate() method should be implementated to invoke super.validate(), assigning the result to a local variable so that it can be compared with null, similar to the following example:

```
public ActionErrors validate(ActionMapping mapping,
HttpServletRequest request) {
ActionErrors errors = super.validate(mapping, request);
if (errors == null){
errors = new ActionErrors();
}
// Code to add additional errors.
return errors;
}
```

**Example**

The following method would be flagged because it does not invoke super.validate():

```
public class MyActionForm extends ActionForm {
public ActionErrors validate(ActionMapping mapping, HttpServletRequest
request) {
ActionErrors errors = new ActionErrors();
errors.add(ActionErrors.GLOBAL_ERROR, new ActionError("error.error"));

...

return errors;
}
}
```

**Invoke Synchronized Method In Loop**

**Severity:** Low

**Summary**

Don't invoke a synchronized method within a loop.

**Description**

This audit rule looks for invocations of methods that have been marked as being "synchronized" that occur within a loop. Synchronization is relatively expensive, so such calls should be made outside the loop if possible.

**Example**

Given a method defined as follows:

```
public synchronized void recomputeCaches()
{
...
}
```

The following invocation would be flagged as an error:

```
public void repeatedlyInvokeIt()
{
for (int i = 0; i < 10; i++) {
recomputeCaches();
}
}
```

**JNDI Naming Standard**

**Severity:** Medium

**Summary**

JNDI names should follow the specified standard.

**Description**

This rule finds JNDI names (within .xmi files) that do not conform to the specified standard.

**Example**

If the rule were configured to require JNDI names to start with a prefix of "ejb/com/myCompany" and have at least one segment representing a functional area, with "accounting" and "manufacturing" on the list, the name "SystemFacade" in the following JNDI name would be flagged as a violation:

```
<ejbBindings jndiName="ejb/com/myCompany/SystemFacade" ...>
```

**JUnit Framework Checks**

**Severity:** Medium

**Summary**

Check JUnit framework method declarations.

**Description**

JUnit testing framework methods are checked for proper declarations, including spelling. The following checks are performed:
- Check that "suite", "setUp" and "tearDown" are spelled correctly
- Check that "suite", "setUp" and "tearDown" have no parameters
- Check that "suite", "setUp" and "tearDown" are void
- Check that "suite", "setUp" and "tearDown" have the proper visibility
- Check that "setUp" and "tearDown" are not static
- Check that "suite" is static

**Label Naming Convention**

**Severity:** Medium

**Summary**

Label names should conform to the defined standard.

**Description**

This audit rule checks the names of all labels.

**Example**

If the rule were configured to require that labels begin with a lower case "l" and an upper case letter, the following label would be flagged as a violation because it does not begin with a lower case "l":

```
here: while (true) {
...
}
```

**Large Number of Constructors**

**Severity:** Medium

**Summary**

Types should not have too many constructors.

**Description**

This audit rule finds types that have more than the specified number of constructors. Types that exceed this number are likely to be too complex. Consider making some of the constructors more general.

**Example**

If the rule is configured to allow 3 constructors and a class with 6 constructors is found, that class will be flagged as a violation.

**Large Number of Fields**

**Severity:** Medium

**Summary**

Types should not have too many fields.

**Description**

This audit rule finds types that have more than the specified number of fields. Types that exceed this number are likely to be too complex. Consider splitting the class into multiple smaller classes.

**Example**

If the rule is configured to allow 6 fields and a class with 15 fields is found, that class will be flagged as a violation.

**Large Number of Methods**

**Severity:** Low

**Summary**

Types should not have too many methods.

**Description**

This audit rule finds types that have more than the specified number of methods. Types that exceed this number are likely to be too complex. Consider splitting the class into multiple smaller classes.

**Example**

If the rule is configured to allow 20 methods and a class with 114 methods is found, that class will be flagged as a violation.

**Large Number of Parameters**

**Severity:** Low

**Summary**

Methods should not have too many parameters.

**Description**

This audit rule finds methods that have more than the specified number of parameters. Methods that exceed this number are likely to be too complex. Consider moving some of the values and behavior associated with them into a separate class.

**Example**

If the rule is configured to allow 4 parameters and a method with 12 parameters is found, that method will be flagged as a violation.

**Large Number of Switch Statement Cases**

**Severity:** Medium

**Summary**

Switch statements should not have more than 256 case clauses.

**Description**

This audit rule looks for switch statements that have more than 256 case clauses. Some processors have special support for switch statements and some JITs will take advantage of such instructions when there are few enough cases.

**Example**

If a switch statement with more than 256 case clauses is found, it will be flagged as a violation.

**Lazily Initialize Singletons**

**Severity:** Medium

**Summary**

Singleton objects should be initialized as late as possible.

**Description**

This audit rule looks for singleton classes in which the singleton object is initialized earlier than necessary.

**Example**

The following static field declaration would be flagged because the field is initialized in the declaration rather than in the getInstance() method:

```
private static MySingleton UniqueInstance = new MySingleton();

public static MySingleton getInstance()
{
return UniqueInstance;
}
```

**Line Length**

**Severity:** Low

**Summary**

Lines should not be too long.

**Description**

This audit rule checks for lines that are longer than a specified number of characters. By default, each tab character is counted as four spaces.

**Example**

If the rule were configured to allow lines of up to 120 character and a line were found that contained 253 characters, that line would be flagged as a violation.

**Local Declarations**

**Severity:** Low

**Summary**

Verification that local variable declarations follow a specified style of coding.

**Description**

This audit rule finds source in which local variable declarations do not follow a specified style of coding. This includes the order of local variables defined in a block of code, whether or not each variable is explicitly initialized, and whether or not each variable is declared in a separate statement.

**Example**

If the rule is configured to require that local variables be declared at the beginning of the method, then the following variable declaration would be flagged as a violation:

```
public boolean equals(Object object)
{
if (!(object instanceof OrderedSet)) {
return false;
}
int thisSize = getSize();
...
}
```

**Local Variable Naming Convention**

**Severity:** Medium

**Summary**

Local variable names should conform to the defined standard.

**Description**

This audit rule checks the names of all local variables (parameters and temporary variables).

**Example**

If the rule were configured to only allow local variables to begin with a lower case letter, the following would be flagged as a violation because it starts with an underscore:

```
int _count;
```

**Log Exceptions**

**Severity:** Medium

**Summary**

Exceptions that are caught should be logged.

**Description**

This audit rule checks for caught exceptions that are not logged.

**Example**

The following catch clause would be flagged as a violation because the exception is not logged:

```
try {
...
} catch (Exception exception) {
// Exceptions should never be ignored like this
}
```

**Log Forging**

**Severity:** High

**Summary**

User input might be getting used to write directly to a log.

**Description**

Log Forging occurs when user input is printed directly to a log or as part of a log.

To detect violations, this audit rule searches the code for logging statements such as `logger.log(..)` and traces where the logging string could have come from. In cases where the source of the path is user input, such as data from a servlet request, `javax.servlet.ServletRequest.getParameter(java.lang.String)`, or from a SWT Text widget, `org.eclipse.swt.widgets.Text.getText()`, a violation is created.

These two sets of methods, the locations where tainted user data can come from and the methods used to create paths, are editable by the user. If methods are missing that are in a common package (such as `java.lang.*`), please let CodePro support know.

**Security Implications**

When a malicious user can enter information directly into the log, the application logging utility can become compromised.

**Example**

The invocation of `log(..)` would be flagged as a violation since it uses the user name information passed from a servlet request:

```
ServletRequest servletRequest = ...;
Logger logger = ...;
Level level = ...;
String userName = servletRequest.getParameter("userName");
String logMessage = "User input the following user name: " + userName;
logger.log(level,logMessage);
```

**Log Level**

**Severity:** Medium

**Summary**

Only those log levels that are allowed should be used.

**Description**

This audit rule checks for used of log levels that are not allowed.

**Example**

If the rule were configured to not allow the use of the ALL level, the following invocation would be flagged as a violation:

```
logger.log(Level.ALL, "This will always be logged");
```

**Log Level**

**Severity:** Medium

**Summary**

Only those log levels that are allowed should be used.

**Description**

This audit rule checks for used of log levels that are not allowed.

**Example**

If the rule were configured to not allow the use of the DEBUG level, the following invocation would be flagged as a violation:

```
logger.debug("This is debugging output");
```

**Loop Variable Naming Convention**

**Severity:** Medium

**Summary**

Variables declared in for loops should conform to the defined standard.

**Description**

This audit rule checks the names of all integer-valued loop variables to see whether they are on a list of approved names.

**Example**

If the rule were configured to only allow loop variable names of "i", "j" and "k", the following loop variable would be flagged as a violation:

```
for (int index = 0; index < array.length; index++) {
...
}
```

**Loss of Precision in Cast**

**Severity:** Medium

**Summary**

Casting to a lower precision type can cause loss of data.

**Description**

This audit rule checks for places where one numeric type is being cast to another type of lower precision than the first. Doing so can result in a loss of data, which is generally not desirable.

**Example**

Given a declaration of the form:

```
double oneThird = 1.0 / 3.0;
```

The following expression would be flagged as a violation:

```
(float) oneThird
```

**Manipulation with XPath**

**Severity:** High

**Summary**

Request parameters and other tainted data should not be passed into methods creating an XPath expression without sanitizing.

**Description**

This rule violates usage of an unvalidated user input as a part of string used in creating an XPath expression.

**Security Implications**

Only trusted data should be used in an xpath expression, otherwise an attacker can perform an xpath injection, potentially bypassing the security system of an application and accessing sensitive data.

**Example**

The following code uses user input directly as an XPath expression:

```
String query = req.getParameter("query");
XPathFactory factory = XPathFactory.newInstance();
XPath xpath = factory.newXPath();
XPathExpression expr = xpath.compile(query);
```

**Message Beans**

**Severity:** Medium

**Summary**

Message Beans should be properly defined.

**Description**

This audit rule finds problems with Message Beans:
- Message bean should be declared public
- Message bean should not be declared as abstract or final
- Message bean name should end with 'Bean'
- Message bean not implement a finalize() method
- ejbCreate() method should be declared public
- ejbCreate() method should not be declared as static or final
- ejbCreate() method should be declared void
- ejbCreate() method should not have any arguments
- ejbCreate() method should not throw any exceptions
- Message bean should implement an ejbCreate() method
- Message bean should implement a no-argument constructor
- ejbRemove() method should be declared public
- ejbRemove() method should not be declared as static or final
- ejbRemove() method should be declared void
- ejbRemove() method should not have any arguments
- ejbRemove() method should not throw any exceptions
- Message bean should implement an ejbRemove() method
- onMessage() method should be declared public
- onMessage() method should not be declared as static or final
- onMessage() method should be declared void
- onMessage() method should have a single javax.jms.Message parameter
- onMessage() method should not throw any exceptions
- Message bean should implement an onMessage() method

**Message Document Must Have Version Attribute**

**Severity:** Medium

**Summary**

The complex type of the message document must contain a "schemaVersion" attribute.

**Description**

This audit rule looks for complex types of a message document that do not have a "schemaVersion" attribute.

**Example**

The following entry should be present in the message document:

```
<attribute
name="schemaVersion"
type="decimal"
use="required"
fixed="1.0"/>
```

**Method Chain Length**

**Severity:** Low

**Summary**

Method invocations should not be chained into a long line.

**Description**

This audit rule looks for places where multiple method invocations are chained together into an overly long chain. There are two problems with long chains of method invocations. The most obvious is that they can make the code difficult to follow. An even more serious problem is that they often point out places where information has not been well encapsulated.

**Example**

```
Color roofColor = party.getWorkAddress().getStructure().getRoof().getColor();
```

**Method Could Be Final**

**Severity:** Medium

**Summary**

Methods that are not supposed to be overridden should be declared `final`.

**Description**

This audit rule looks for non-final, non-abstract method declarations that are declared in a non-

final class and are not overridden in subclasses. Such methods should be declared `final` so that subclasses cannot maliciously redefine the behavior of the method.

**Security Implications**

A malicious user can redefine the behavior of the method so that it affects the whole class in an unexpected or insecure way.

**Example**

`checkSecurity()` would be flagged as a violation because it is declared in a non-final class, is not abstract, and is not overridden in subclasses:

```
public class SecureDataAccess {
protected void checkSecurity() {
...
}
}
```

**Method Invocation in Loop Condition**

**Severity:** Medium

**Summary**

Methods should not be invoked in a loop condition.

**Description**

This audit rule looks for places where a method is invoked as part of a loop condition. Unless the method returns a different value each time it is called, placing the method invocation in the loop condition will force it to be executed at least as many times as the loop body. You can often improve the performance of your code by moving the invocation before the loop.

**Example**

The invocation of the method size() would be flagged as a violation in the following loop:

```
for (int i = 0; i < list.size(); i++) {
...
}
```

**Method Javadoc Conventions**

**Severity:** Medium

**Summary**

All methods should have a Javadoc comment associated with them.

**Description**

This audit rule checks for the existence of a Javadoc comment for each method. In addition, it checks that each Javadoc comment includes a @param tag for each parameter (and none for non-parameters), a @return tag if the method has a return type other than void (and not if the return type is void), and a @throws tag for each explicitly declared exception (and none for exceptions that are not declared). It also checks for the use of the obsolete @exception tag.

**Example**

The Javadoc for the following method would be flagged three times as a violation, twice for missing @param tags and once for a missing @return tag:

```
/**
* Return the sum.
*/
public int sum(int x, int y)
{
...
}
```

**Method Naming Convention**

**Severity:** Medium

**Summary**

Method names should conform to the defined standard.

**Description**

This audit rule checks the names of all methods.

**Example**

If the rule were configured to only allow method names that begin with a lower case letter, the following method declaration would be flagged as a violation because the method name begins with an upper case letter:

```
public static Singleton GetInstance()
```

**Method Parameter Naming Convention**

**Severity:** Medium

**Summary**

Method parameter names should conform to the defined standard.

**Description**

This audit rule checks the names of all method parameters.

**Example**

If the rule were configured to only allow parameter names that begin with a lower case letter, the parameter in the following method declaration would be flagged as a violation because it begins with an underscore:

```
public void setCount(int _count)
```

**Method Should Be Private**

**Severity:** Medium

**Summary**

Method should be private if it is not member of an interface and it doesn't override method of superclass and isn't overriden by subtype.

**Description**

This audit rule looks for non-private method declarations in subclasses of a specified class that are declared in a class and but are neither overridden in subclasses nor are overrides of methods of a superclass.

**Security Implications**

In weakly protected environments (like applets) a malicious developer can create code that can use non-private methods and thus access data that otherwise should not be accessible.

**Example**

If MyClass is added to the list of weakly protected classes, the following method declaration whould be marked as violation because this method is non-private:

```
public class MyClass {
public void func() {
...
}
}
```

**Method Should Be Static**

**Severity:** Medium

**Summary**

Methods that do not access any instance state or instance methods should be static.

**Description**

This audit rule checks for methods that do not access any instance state or instance methods. Such methods should be made static. Note, however, that this method does not find methods that only call instance methods that should also be static, so changing some of the methods to static methods might cause additional methods to then be flagged.

**Example**

The following method would be flagged as a violation because it does not reference any instance specific state:

```
public int getDefaultSize()
{
return 256;
}
```

**Mime Encoding Method Usage**

**Severity:** Low

**Summary**

Don't use specific mime encoding methods.

**Description**

Don't use specific mime encoding methods.

**Minimize Scope of Local Variables**

**Severity:** Medium

**Summary**

Declare variables so that they have as small a scope as possible.

**Description**

This rule looks for variables whose scope is too broad. If a variable is declared without an initializer it is probably declared too early, and will be flagged. While-loops that could be converted to for-loops, reducing the scope of the iteration variable, are also detected.

**Example**

In the following method, the variable "element" could be defined in the body of the loop, and the while loop could be converted to a for loop so that the loop variable "index" could be more limited in scope:

```
public int sum(int[] array)
{
int index, element;

index = 0;
while (index < array.length) {
element = array[i];
sum = sum + element;
index = index + 1;
}
}
```

**Mismatched Notify**

**Severity:** Medium

**Summary**

Do not use `notify()` or `notifyAll()` methods without holding locks.

**Description**

This audit rule looks for invocations of `notify()` or `notifyAll()` methods without holding a lock on the object.

**Security Implications**

Invoking `notify()` or `notifyAll()` without holding a lock can lead to throwing an `IllegalMonitorStateException`.

**Example**

The following invocation of the `notify()` method will be marked as a violation because the container method is not `synchronized`:

```
public void method() {
Notify();
}
```

**Mismatched Wait**

**Severity:** Medium

**Summary**

Do not use `wait()` method without holding locks.

**Description**

This audit rule looks for invocations of the `wait()` method without holding a lock on the object.

**Security Implications**

Invoking `wait()` without holding a lock can lead to throwing an `IllegalMonitorStateException`.

**Example**

The following invocation of the `wait` method will be marked as a violation because the container method is not `synchronized`:

```
public void method(){
wait();
}
```

**Missing Application Context File**

**Severity:** Medium

**Summary**

XML files referenced in 'contextConfigLocation' attribute of <context-param> section in web.xml configuration file should exist.

**Description**

This audit rule looks for references to application context configuration files that do not exist. The application context configuration files are referenced in the contextConfigLocation attribute of a <context-param> section in the web.xml configuration file.

**Example**

Given the following content in a web.xml file:

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>
/WEB-INF/dataAccessContext.xml
/WEB-INF/applicationContext.xml
</param-value>
</context-param>
```

The file /WEB-INF/dataAccessContext.xml would be flagged if the file does not exist, as would the file /WEB-INF/applicationContext.xml.

**Missing Assert in JUnit Test Method**

**Severity:** Medium

**Summary**

JUnit tests should include at least one assertion.

**Description**

This audit rule flags any JUnit test method that does not contain any assertions. Missing Assertions usually indicate weak or incomplete test cases. Using assert with messages provide a clear idea of what the test does.

**Example**

The following test case would be flagged as a violation because it does not contain any assertions:

```
public class MissingAssert extends TestCase{
public void testSomething(){
Foo f = findFoo();
f.doWork();}}
```

**Missing Bean Description**

**Severity:** Low

**Summary**

Every bean declaration should have a description.

**Description**

This audit rule looks for bean declarations that do not include a description. The advantage of using the description element is that it is easy for tools to pick up the description from this element.

**Example**

Bean declarations will be flagged unless they contain a description element such as the following:

```
<beans>
<description>
...
</description>
...
</beans>
```

**Missing Block**

**Severity:** Medium

**Summary**

A single statement should never be used where a block is allowed.

**Description**

This audit rule checks for statements that control the execution of another statement (do, for, if or while) to ensure that the statement being controlled is always a block.

**Example**

```
if (color == null)
color = getDefaultColor();
```

**Missing Catch of Exception**

**Severity:** Medium

**Summary**

The methods `doGet()`, `doPost()`, and others should catch ALL `Exception`s.

**Description**

This audit rule violates HttpServlet serving methods that do not catch all exceptions.

**Security Implications**

Exceptions thrown from a servlet usually end with a stack trace printed to the end user. This stack trace may contain details of your system's architecture that provide valuable information for the attacker.

**Example**

The following code would be flagged as a violation because it does not catch Exceptions, including runtime ones, that may possibly be thrown from calcDefaultResponse() method:

```
public class MissingCatchOfExceptionTest extends HttpServlet {
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
try {
resp.getWriter().write(calcDefaultResponse());
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

**Missing Class in web.xml**

**Severity:** Medium

**Summary**

A class referenced in a web.xml file is not defined.

**Description**

This audit rule checks the values of those attributes in a web.xml file whose values are supposed to be the fully qualified name of a class to ensure that the classes are defined.

**Example**

In the following entry

```
<controller className="your class here" .../>
```

The value of the "className" attribute would be flagged.

**Missing Constants In Switch**

**Severity:** Medium

**Summary**

Switch statements should include all possible enumeration constants.

**Description**

This audit rule checks for the existence of switch statements whose case labels are constants declared by an enum but which do not include all of the declared constants.

**Example**

Given the following declarations:

```
public enum PopcornSize {MEDIUM, LARGE, EXTRA_LARGE};

private PopcornSize size;
```

the following switch statement would be flagged as a violation because it does not contain a case label for EXTRA_LARGE:

```
switch (size) {
case MEDIUM:
promptForUgradeToLarge();
break;
case LARGE:
promptForDrinksAndCandy();
}
```

**Missing Default in Switch**

**Severity:** Medium

**Summary**

Every switch statement should have a default clause.

**Description**

This audit rule checks for the existence of a default case within every switch statement.

**Example**

The following switch statement would be flagged as a violation because it does not contain a "default" case label:

```
switch (accountType) {
case CHECKING_ACCOUNT:
balance = ((CheckingAccount) account).getCheckingBalance();
break;
case SAVINGS_ACCOUNT:
balance = ((SavingsAccount) account).getSavingsBalance();
}
```

**Missing Error Page**

**Severity:** Medium

**Summary**

Error page should handle any exception in a web application.

**Description**

This audit rule violates the web-app configuration file that does not declare a default error page, i.e. error page that catches all exceptions.

**Security Implications**

Exceptions thrown from a servlet usually end with a stack trace printed to the end user. This stack trace may contain details of your system's architecture that provide valuable information for the attacker. To prevent this, add a default error that handles any Throwable.

**Example**

The following web application declaration would be flagged as a violation because it does not handle any Throwable, including errors:

```
<web-app>
<error-page>
<exception-type>java.io.IOException</exception-type>
<location>/error.jsp</location>
</error-page>
</web-app>
```

**Missing Image File**

**Severity:** Medium

**Summary**

Referenced image files should exist.

**Description**

This audit rule checks for the references within a plugin.xml file to image files that do not exist.

**Example**

If the file `icons/view16/ourView.gif` does not exist, the following reference to it would be flagged:

```
<view icon="icons/view16/ourView.gif" .../>
```

**Missing Message**

**Severity:** Medium

**Summary**

Messages referenced in code should exist in property files.

**Description**

This audit rule looks for subclasses of `org.eclipse.osgi.util.NLS` and verifies that there is a key in the corresponding `messages.properties` file for each of the fields defined in it. It also looks for invocations of the older getString() method to ensure that there is a key corresponding to the argument passed in to that method (if the run-time value can be determined at compile time).

**Example**

If a subclass had a field like the following:

```
public static String Externalized_String;
```

and the `messages.properties` file did not contain a key of `Externalized_String`, then a violation would be created.

**Missing Message in Assert**

**Severity:** Medium

**Summary**

Assertions should have messages.

**Description**

This audit rule looks for invocations of the assert and fail methods defined in junit.framework.Assert which do not have a message. A message should be added so that the framework can provide better information about why the test failed.

**Example**

The following invocation would be flagged as a violation:

```
assertEquals(expectedCount, actualCount);
```

**Missing Namespace Grouping Identifiers**

**Severity:** Medium

**Summary**

One or more grouping identifiers should be used to further clarify the namespace.

**Description**

This audit rule looks for namespace URIs that do not contain either a service name as a part of a namespace or one of the list of grouping identifiers. Grouping identifiers should be used to further clarify the namespace. This could be the service name or some other grouping that would separate it from elements used elsewhere.

**Example**

The following service declaration does not use a service name as a part of its `xmlns:tns` and would thus be marked as a violation:

```
<definitions
name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
```

```
xmlns:tns="http://example.com/stockquote.wsdl"
xmlns:xsd1="http://example.com/stockquote.xsd"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
...
</definitions>
```

**Missing Namespace Version**

**Severity:** Medium

**Summary**

At the end of the namespace, the version of the WSDL or schema must be specified.

**Description**

Version numbers must have a major number specified such as v2. The second and third version numbers are left off of service artifact naming because fixes/minor releases generally do not affect the service interface. The 'v' must be lower case.

**Example**

The following namespaces would be flagged as a violation because they do not include a version number:

```
http://wsdl.example.com/wsdl/product/ProductService
http://schema.example.com/AutoValidationService
```

They should be replaced by something like the following:

```
http://wsdl.example.com/wsdl/product/ProductService/v1
http://schema.example.com/AutoValidationService/v2
```

**Missing Or Misplaced Manifest Version**

**Severity:** High

**Summary**

Every manifest file must begin with the manifest version.

**Description**

This audit rule finds manifest files that do not contain an attribute named "Manifest-Version", or in which the manifest version is not the first attribute listed in the file.

**Missing Or Misplaced Section Name**

**Severity:** High

**Summary**

Every section in a manifest file must begin with a name.

**Description**

This audit rule finds manifest file sections that do not contain an attribute named "Name", or in which the name is not the first attribute listed in the file.

**Missing reset() Method**

**Severity:** Medium

**Summary**

Struts `ActionForm` subclasses should implement the `reset()` method.

**Description**

This audit rule finds `ActionForm` subclasses that do not implement the `reset()` method.

**Security Implications**

If some fields are not reset, they can have old values which can be used by an attacker.

**Example**

The following class declaration would be flagged as a violation because it does not declare a `reset()` method:

```
public class SomeForm extends ActionForm { }
```

**Missing static method in non-instantiable class**

**Severity:** Medium

**Summary**

Non-instantiable classes should have at least one static method.

**Description**

If a class has been made non-instantiable by making all constructors private, it should define at least one non-private static method, otherwise the class will be unusable.

**Example**

The following would be flagged as a violation:

```
public class Foo {
private Foo() {
}
}
```

**Missing Update in For Statement**

**Severity:** Medium

**Summary**

Every for statement should have an update clause.

**Description**

This audit rule checks for the existence of an update clause within every for statement. If a for statement does not require an update clause it should be replaced by a while statement.

**Example**

The following for statement would be flagged as a violation:

```
for (Iterator i = set.iterator(); i.hasNext(); ) {
Object element = i.next();
```

```
...
}
```

**Missing Validator Name**

**Severity:** Medium

**Summary**

All validators used in validation logic should be declared in `validators.xml` configuration file.

**Description**

This audit rule finds and violates the usages of validators whose `type` is not declared in `validators.xml`.

**Security Implications**

The absence of a declaration of a validator used is a sign of a faulty approach to secure validation all other the application. Validation logic should be kept up to date.

**Example**

The following `customBarFieldValidation` validator usage whould be marked as violation if the corresponding validator is not declared in `validators.xml` configuration file:

```
<validators>
<field name="bar">
<field-validator type="customBarFieldValidation">
<message>You must enter a value for bar.</message>
</field-validator>
</field>
</validators>
```

**Misspelled Method Name**

**Severity:** Medium

**Summary**

Methods with incorrectly spelled names do not override the superclass method.

**Description**

This rule detects small differences in spelling between methods defined in two different types in a hierarchy where both methods have the same parameter lists. Such methods may be intended to be identical but only the superclass method will be invoked.

**Modifier Order**

**Severity:** Medium

**Summary**

Modifiers should always appear in the standard order.

**Description**

This audit rule checks the order in which the modifiers were declared and reports any deviation from the standard (Sun defined) order.

**Example**

```
final static public int ANSWER = 42;
```

**More Than One Logger**

**Severity:** Medium

**Summary**

Use one shared logger instance per class.

**Description**

Using more than one logger instance per class is sometimes considered a bad coding style. This rule will create a violation for each class that has more than one logger instance defined.

**Example**

The following class declaration would be flagged as a violation because it contains more than one logger:

```
public class Something {
private static final Log errorsLogger = LogFactory.getLog(Something.class);
private static final Log accessLogger = LogFactory.getLog(Something.class);
}
```

**Multiple Return Statements**

**Severity:** Medium

**Summary**

Methods should have a single return statement.

**Description**

This audit rule looks for methods that contain multiple return statements (or constructors or void methods that contain any return statements). Such methods are generally too complex and need to be rewritten or refactored in order to make their logic easier to understand.

**Example**

The following method would be flagged:

```
public int returnCode(String str)
{
if (str == null) {
return 0;
} else if ("ABC".equals(str)) {
return 1;
} else if ("123".equals(str)) {
return 2;
} else {
return -1;
}
}
```

**Multiplication Or Division By Powers of 2**

**Severity:** Medium

**Summary**

Do not multiply or divide by powers of 2.

**Description**

This audit rule checks for multiplication or division by powers of 2. "*" and "/" are an expensive operations. The shift operator is faster and more efficient.

**Example**

The following expression would be flagged as a violation:

```
list.size() * 2;
```

**Mutability Of Arrays**

**Severity:** Medium

**Summary**

Do not return internal arrays from non-private methods.

**Description**

This audit rule flags unsafe set or get of an internal array field, this includes flagging: 1) internal array returns from a non-private method and 2) setting an internal array with parameter given through a non-private method.

**Security Implications**

With arrays, it is safer to make a copy before the set or return. This way, the internal data cannot be manipulated outside of the class.

**Example**

Given the following field declaration:

```
int[] integerArray = ...;

public int[] getIntegerArray() {
return integerArray;
}
```

**Mutable Constant Field**

**Severity:** Medium

**Summary**

Disallows public static final mutable type fields.

**Description**

Checks that public static final fields are either default immutable types like java.lang.Integer or are user-specified immutable types, declared through the audit rule preferences.

**Security Implications**

Public static final mutable fields are accessible and changeable from all points in the application and thus are targets by malicious users.

**Example**

The following would be flagged:

```
public static final MutableType mutableType;
```

**Nested Synchronized Calls**

**Severity:** Medium

**Summary**

Invoking one synchronized method of an object from another synchronized method of the same object affects the performance of an application.

**Description**

This audit rule looks for invocations of a synchronized method from another synchronized method in the same class.

**Security Implications**

Such calls both affect the performance of an application and indicate a poorly designed synchronization aspect of the code, which usually results in synchronization errors that could be exploited to create unexpected states of an application.

**Example**

The following code would be flagged as a violation because it invokes one synchronized method of an object from another synchronized method of the same object:

```
public class SyncDataSource {
public synchronized Object getData() {
return internalGetData();
}
private synchronized Object internalGetData() {
...
}
}
```

**Never Use the Identifier in equals() or hashCode()**

**Severity:** High

**Summary**

Never ever use the database identifier in the equals() and hashCode() methods, because a transient object doesn't have an identifier value.

**Description**

This audit rule looks for uses of the identifier field in the implementation of either the equals() or hashCode() methods. A transient object doesn't have an identifier value because Hibernate will only assign a value when the object is saved. To implement equals() and hashCode(), use a unique business key, that is, compare a unique combination of class properties.

**Example**

The equals() method would be flagged because it uses the identifier property.

```
public class Message {
private Long id;
private String text;
private Message nextMessage;

...

public boolean equals(Object obj) {
if (obj instanceof Message) {
Message message = (Message)obj;
if (message.getId() == getId()) {
return true;
```

```
}
}
return false;
}
}
```

**Next method invoked without hasNext method**

**Severity:** Medium

**Summary**

Do not invoke the `next` method if you do not invoke `hasNext` method before that because NoSuchElementException can be thrown.

**Description**

This rule looks for places where the `next` method is invoked without or before the `hasNext` method.

**Security Implications**

If the `next` method is invoked without first invoking the `hasNext` method, in loop for example, a NoSuchElementException may be thrown.

**Example**

The following invocation of the `next` method will be flagged as a violation because the `hasNext` method is not invoked:

```
public void myMethod(Collection myList)
{
.......
Iterator iter = myList.iterator();
for (int i = 0; i < 10; i++) {
iter.next();
}
}
```

**No Abstract Methods**

**Severity:** Medium

**Summary**

Classes should not be declared abstract unless they define abstract methods.

**Description**

This audit rule finds classes that are declared to be abstract but that do not define any abstract methods.

**Example**

The following class declaration would be flagged as a violation because it does not define any abstract methods:

```
public abstract class Widget
{
}
```

**No Action For Validation**

**Severity:** Medium

**Summary**

Abandoned validation file is a sign that validation of the actions is not properly handled.

**Description**

A validation file without a corresponding action could be a result of an action renaming, when developer forgot to rename a validation file.

**Security Implications**

It indicates problems in the validation of the actions, that should be treated.

**Example**

When file `AddItem-validation.xml` is found with no corresponding action in `struts.xml`, the violation is created. The proper action declaration in this case would look like this:

```
<struts>
...
```

```
<package name="sample" namespace="/sample" extends="struts-default">
<action name="AddItem" class="my.sample.AddItemAction">
<result>/pages/AddItem.jsp</result>
</action>
</package>
...
</struts>
```

**No DB Driver Loading**

**Severity:** Medium

**Summary**

Do not reference database driver loading classes.

**Description**

This audit rule checks for classes that are used to connect to databases that should not be used within the code.

**Example**

If the class java.sql.DriverManager is on the list of disallowed classes, then the following method invocation would be flagged as a violation:

```
DriverManager.getDrivers();
```

**No Explicit Exit**

**Severity:** Medium

**Summary**

The methods System.exit(int), Runtime.exit(int), and Runtime.halt(int) should not be invoked.

**Description**

This audit rule checks for invocations of either of the exit methods, System.exit(int) and Runtime.exit(int), or Runtime.halt(int).

**Example**

The following invocation would be flagged as a violation:

```
System.exit(0);
```

**No Explicit This Use in EJB's**

**Severity:** Medium

**Summary**

Don't use the keyword "this" in EJB classes.

**Description**

This audit rule checks for EJB classes (subclasses of EnterpriseBean) that explicitly reference the bean using the keyword "this".

**Example**

The following return statement would be flagged as a violation:

```
return this;
```

**No Instance Fields In Portlets**

**Severity:** Medium

**Summary**

Portlets should not declare any instance fields.

**Description**

This audit rule checks for the declaration of instance fields in portlets.

**Example**

The following field would be flagged as a violation if it occurred within a portlets:

```
private int useCount;
```

**No Instance Fields In Servlets**

**Severity:** Medium

**Summary**

Servlets should not declare any instance fields.

**Description**

This audit rule checks for the declaration of instance fields in servlets.

**Example**

The following field would be flagged as a violation if it occurred within a servlet:

```
private int useCount;
```

**No Public Members**

**Severity:** Medium

**Summary**

Classes should not be declared public unless they define public members.

**Description**

This audit rule finds classes that are declared to be public but that do not define any public members or at least one protected constructor. Consider restricting such classes to package scope.

**No Run Method**

**Severity:** Medium

**Summary**

Subclasses of Thread should implement the run() method.

**Description**

Subclasses of Thread should implement the run() method so that they will have the behavior for which they were created.

**Example**

The following class will be flagged as a violation because it does not implement a run() method even though it subclasses java.lang.Thread:

```
public class SuperThread extends Thread
{
}
```

**No Set-up in Constructors**

**Severity:** Medium

**Summary**

TestCase constructors should not include any set-up code.

**Description**

This audit rule checks for constructors defined in test cases that do more than invoke the superclass' constructor. Set-up should be performed in either the setUp() method or the accessor method for one of the test fixtures.

**Example**

The following constructor would be flagged as a violation if it occurred within a test case class:

```
public MyTestCase(String testName)
{
super(testName);
employee = new Employee("Jane Doe");
}
```

**No Such Field For Validation**

**Severity:** Medium

**Summary**

Abandoned field validation is a sign that validation of the actions is not properly handled.

**Description**

A field validation declared without a corresponding field in the action class could be a result of a field being renamed, when the developer forgot to replace the field name in a validation file.

**Security Implications**

It indicates problems in the validation of the actions, that should be treated.

**Example**

Let there be a Struts form class, `PostForm`, that declares two fields, `title` and `message`. The following entry in the file `PostForm-validation.xml` will lead to a violation because it validates a non-existent field:

```
<field name="name">
<field-validator type="requiredstring">
<message>You must enter a name</message>
</field-validator>
</field>
```

**No Timestamp In Client Request**

**Severity:** Medium

**Summary**

Websphere web service client should add timestamp to all produced messages.

**Description**

This audit rule looks for declarations of Websphere web service client extensions and violates the declarations which do not declare adding the timestamp to a request message.

**Security Implications**

A timestamp in the message allows the receiver to make a decision about trusting a message

based on its freshness. If an attacker intercepts a message and modifies it, an outdated timestamp may be enough to indicate the problem.

**Example**

The following code does not declare a timestamp as a part of a produced message via `<addCreatedTimeStamp>` configuration option, thus the whole `<securityRequestSenderServiceConfig>` declaration is marked as violation:

```
<securityRequestSenderServiceConfig>
<integrity>
<references part="body"/>
<references part="timestamp"/>
</integrity>
</securityRequestSenderServiceConfig>
```

**No Timestamp In Client Response**

**Severity:** Medium

**Summary**

Websphere web service client should require a timestamp in all received messages.

**Description**

This audit rule looks for declarations of Websphere web service client extensions and violates the declarations which do not require a timestamp in a response message.

**Security Implications**

A timestamp in the message allows the receiver to make a decision about trusting a message based on its freshness. If an attacker intercepts a message and modifies it, an outdated timestamp may be enough to indicate the problem.

**Example**

The following code does not declare a timestamp as a part of a response message via `<addReceivedTimeStamp>` configuration option, thus the whole `<securityResponseReceiverServiceConfig>` declaration is marked as violation:

```
<securityResponseReceiverServiceConfig>
<requiredIntegrity>
```

```
<references part="body"/>
<references part="timestamp"/>
</requiredIntegrity>
</securityResponseReceiverServiceConfig>
```

**No Timestamp In Server Request**

**Severity:** Medium

**Summary**

Websphere web service Server should require a timestamp in all consumed messages.

**Description**

This audit rule looks for declarations of Websphere web service Server extensions and violates the declarations which do not require the timestamp in a request message.

**Security Implications**

A timestamp in the message allows the receiver to make a decision about trusting a message based on its freshness. If an attacker intercepts a message and modifies it, an outdated timestamp may be enough to indicate the problem.

**Example**

The following code does not require a timestamp as a part of a consumed message via `<addReceivedTimestamp>` configuration option, thus the whole `<securityRequestReceiverServiceConfig>` declaration is marked as violation:

```
<securityRequestReceiverServiceConfig>
<requiredIntegrity>
<references part="body"/>
<references part="timestamp"/>
</requiredIntegrity>
</securityRequestReceiverServiceConfig>
```

**No Timestamp In Server Response**

**Severity:** Medium

**Summary**

Websphere web service Server should require a timestamp in all produced messages.

**Description**

This audit rule looks for declarations of Websphere web service Server extensions and violates the declarations which do not add a timestamp to a response message.

**Security Implications**

A timestamp in the message allows the receiver to make a decision about trusting a message based on its freshness. If an attacker intercepts a message and modifies it, an outdated timestamp may be enough to indicate the problem.

**Example**

The following code does not declare a timestamp as a part of a response message via `<addCreatedTimestamp>` configuration option, thus the whole `<securityResponseSenderServiceConfig>` declaration is marked as violation:

```
<securityResponseSenderServiceConfig>
<integrity>
<references part="body"/>
<references part="timestamp"/>
</integrity>
</securityResponseSenderServiceConfig>
```

**No Validation Message**

**Severity:** Medium

**Summary**

Validation methods should indicate why the input is invalid.

**Description**

A validation method can signal invalid input either by invoking setValid with an argument of false, or by throwing a ValidatorException. If setValid is invoked, the method addMessage should be invoked to pass a message back to the user explaining why the input is not valid. If an exception is thrown, it should have a message that serves the same purpose.This audit rule looks

for validation methods that do not either invoke addMessage() or include a message when creating the ValidatorException.

**Example**

The following validator will be flagged as violation because it neither invokes UIInput.addMessage() nor throws a ValidatorException.

```
public class MyValidator implements Validator {
public void validate(FacesContext context, UIComponent comp, Object value) {
System.out.println("Validating value " + value);
}
}
```

**Non Static Logger**

**Severity:** Medium

**Summary**

Loggers should be declared both `static` and `final`.

**Description**

Loggers should be declared both `static` and `final` so that every instance of a class shares the common logger object.

**Example**

The following class declaration would be flagged as a violation because it does not declare its logger `static final`:

```
public class Something {
private final Log errorsLogger = LogFactory.getLog(Something.class);
...
}
```

**Non-atomic File Operations**

**Severity:** Medium

**Summary**

Checking a file for existence and then writing into it is a non-atomic operation, the assumption about it being atomic may fail, leading to unexpected consequences.

**Description**

This audit rule looks for conditional decisions made on the basis of an invocation of `java.io.File.exists()`.

**Security Implications**

Calling `exists()` and then performing some file operation based on the result of this call is a non-atomic operation, i.e. the file state of existence can change in the gap of time between `exists()` call and the subsequent actions. This may result in an unexpected behavior of an application that could possibly compromise its security.

**Example**

The following code would be flagged as a violation because it makes a file writing decision based on the result of `exists()` call:

```
File lock = new File(".lock");
if (!lock.exists()) {
lock.createNewFile();
...
lock.delete();
}
```

**Non-blank Final Instance Field**

**Severity:** Medium

**Summary**

Final instance fields should be blank.

**Description**

This audit rule finds instance fields that are marked as final and have a value assigned to them in the declaration. If the value of the field is the same for all instances, which it must be in this case, it should be a static field instead. If the value can be different in different instances, then either the field should not be final or the field should be assigned its value in a constructor.

**Example**

The following field declaration would be flagged as a violation:

```
private final int maxItemCount = 64;
```

**Non-case Label in Switch**

**Severity:** Medium

**Summary**

Switch statements should only contain case labels.

**Description**

This audit rule finds labels other that case labels that appear within a switch statement. Such labels are often the result of forgetting to type the keyword "case" rather than an intent to use a labeled statement. If it isn't the result of an accident, having a labeled statement in a switch statement makes the logic much harder to understand because it can easily be mistaken for a case label.

**Example**

The statement labeled "SAVINGS_ACCOUNT" would be flagged as a violation:

```
switch (accountType) {
case CHECKING_ACCOUNT:
balance = ((CheckingAccount) account).getCheckingBalance();
SAVINGS_ACCOUNT:
balance = ((SavingsAccount) account).getSavingsBalance();
}
```

**Non-conforming Backing Bean Methods**

**Severity:** Medium

**Summary**

Verifies that parameters and return types of bean's public methods are those expected by the framework.

**Description**

Public methods of backing beans are called from inside the JSF framework and are expected to have some well-defined signatures. The methods can either be:

- getters/setters of bean properties (to be bound to JSP components or tag attributes)

- event handlers (accepting an instance of an event class and returning nothing)

- action methods (accepting nothing and returning a String)

- validation methods (accepting FacesContext, UIComponent, Object arguments and returningnothing; usually include "validate" prefix in their name)

A method with a signature not matching one of the listed ones will be caught by this rule.

**Example**

The following method will be marked as violation because it conforms to neither of the allowed specs.

```
public class MyBackingBean {
public void violatingMethod(int x, int y) {
...
}
}
```

**Non-private Constructor in Static Type**

**Severity:** Medium

**Summary**

Constructors in classes containing only static members should be private.

**Description**

This audit rule finds non-private constructors in classes containing only static members. There is no value in creating an instance of a type that contains only static members. To prevent such instantiation, ensure that type has a single, no-argument, private constructor and no other constructors.

**Example**

The following class declaration would be flagged as a violation:

```
public class Utilities
{
public static int getSize(List list)
{
...
}
}
```

**Non-protected Constructor in Abstract Type**

**Severity:** Medium

**Summary**

Constructors in abstract classes should be protected.

**Description**

This audit rule finds non-protected constructors in abstract classes. Constructors in an abstract class can only be called from an instantiating subclass. Marking all constructors protected will help indicate this.

**Example**

The constructor in the following class declaration would be flagged as a violation:

```
public abstract class Widget
{
public Widget()
{
...
}
}
```

**Non-serializable Class Declares readObject or writeObject**

**Severity:** Medium

**Summary**

Classes that do not implement Serializable should not declare methods named readObject or writeObject.

**Description**

This audit rule looks for declarations of methods named readObject or writeObject in classes that do not implement the interface java.io.Serializable.

**Example**

The following method would be flagged as a violation:

```
public class NonSerializable
{
private void readObject(ObjectInputStream in)
throws IOException, ClassNotFoundException
{
...
}
}
```

**Non-serializable Class Declares serialVersionUID**

**Severity:** Medium

**Summary**

Classes that do not implement Serializable should not declare a field named serialVersionUID.

**Description**

This audit rule looks for declarations of fields named serialVersionUID in classes that do not implement the interface java.io.Serializable.

**Example**

The following field would be flagged as a violation:

```
public class NonSerializable
{
private static final long serialVersionUID = 0x10F7L;
}
```

**Non-terminated Case Clause**

**Severity:** Medium

**Summary**

Case clauses should never fall through into the following case.

**Description**

This audit rule checks for the existence of either a break, continue, return, or throw statement at the end of each case clause in a switch statement. The lack of either of these statements means that control will fall through to the next case, which is usually not what is intended. It is possible to configure this rule to also accept a user-defined comment (such as "no break") as a signal that the developer knew what was happening.

**Example**

```
switch (accountType) {
case CHECKING_ACCOUNT:
balance = ((CheckingAccount) account).getCheckingBalance();
case SAVINGS_ACCOUNT:
balance = ((SavingsAccount) account).getSavingsBalance();
}
```

**Nonce Not Used**

**Severity:** Medium

**Summary**

<wsse:Nonce> and <wsu:Created> should be used in UsernameToken configuration.

**Description**

This audit rule violates Websphere UsernameToken configurations that do not declare the usage of both <wsse:Nonce> and <wsu:Created> elements.

**Security Implications**

Two optional elements are introduced in the <wsse:UsernameToken> element to provide a countermeasure for replay attacks: <wsse:Nonce> and <wsu:Created>. A nonce is a random value that the sender creates to include in each UsernameToken that it sends. Although using a nonce is an effective countermeasure against replay attacks, it requires a server to maintain a cache of used nonces, consuming server resources. Combining a nonce with a creation

timestamp has the advantage of allowing a server to limit the cache of nonces to a "freshness" time period, establishing an upper bound on resource requirements.

**Example**

The following sample configuration in `ws-security.xml` Websphere configuration file has `com.ibm.ws.wssecurity.config.token.BasicAuth.Nonce` property disabled and would thus be marked as violation:

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wssecurity:WSSecurity xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.etools.webservice.wssecurity="http://www.ibm.com/websphere/apps
erver/schemas/5.0.2/wssecurity.xmi" xmi:id="WSSecurity_1084441805509">
<properties xmi:id="Property_1057972161460"
name="com.ibm.ws.wssecurity.config.token.BasicAuth.Nonce" value="false"/>
<properties xmi:id="Property_1057972161461"
name="com.ibm.ws.wssecurity.config.token.BasicAuth.NonceRequired"
value="false"/>
</com.ibm.etools.webservice.wssecurity:WSSecurity>
```

**Notify method invoked while two locks are held**

**Severity:** Medium

**Summary**

Do not invoke the `notify` or `notifyAll` methods while two locks are being held.

**Description**

This rule looks for places where the code invokes either `notify()` or `notifyAll()` while two locks are held.

**Security Implications**

If this notification is intended to wake up a `wait()` that is holding the same locks, it might deadlock. The wait will only give up one lock and the notify will be unable to get both locks, and thus the notify will not succeed.

**Example**

The following invocation of the `notify` method will be flagged as a violation because two lock are being held.

```
public synchronized void myMethod(Object obj)
{
synchronize (obj) {
obj.notify();
}
}
```

**Null Pointer Dereference**

**Severity:** High

**Summary**

Null pointers should not be dereferenced.

**Description**

This audit rule checks for places where a value that is known to be null is being dereferenced in a way that is guaranteed to produce a NullPointerException.

**Example**

The following code will cause a violation because the variable `o` is known to have a null value:

```
Object o = null;
System.out.println(o.getClass().getName());
```

**Numeric Literals**

**Severity:** Medium

**Summary**

Numeric literals should not appear in code.

**Description**

This audit rule checks for numeric literals within the code that are not included in a user-defined list of acceptable literal values. The list initially contains only the values "-1", "0", and "1".

**Example**

```
int answer = 42;
```

**Obey General Contract of Equals**

**Severity:** Medium

**Summary**

Obey the general contract when overriding equals().

**Description**

The equals() method defined in Object is intended to be overridden by subclasses but each subclass must obey the general contract specified by the superclass. Classes should not use the name "equals" for methods that take any parameters other than a single Object. The body of the method should be coded defensively to accept any class of object as its argument.

**Example**

The style of equals() definition that this rule looks for is this:

```
public boolean equals(Object arg) {
if (this == arg)
return true;
if (!(arg instanceof Foo))
return false;
Foo fooArg = (Foo) arg;
...
}
```

"Foo" is the name of a type, which is either the class that declares this equals() method or an interface that is implemented by that class.

**Obsolete Modifier Usage**

**Severity:** Medium

**Summary**

Obsolete modifiers should not be used.

**Description**

There are a number of modifiers that can validly be specified according to the Java Language Specification, but which Sun strongly discourages the use of. This audit rule checks for places in which those modifiers are used.

**Example**

```
public abstract interface SpecialObject
{
...
}
```

**One Class per Mapping File**

**Severity:** Medium

**Summary**

Use one mapping file per persistent class.

**Description**

This audit rule looks for mapping files declaring mappings for multiple classes. Although it's possible to declare mappings for multiple classes in one mapping file by using multiple <class> elements, the recommended practice (and the practice expected by some Hibernate tools) is to use one mapping file per persistent class.

**Example**

Given a mapping file (*.hbm.xml) containing the following, the second "class" entry would be flagged:

```
<hibernate-mapping>

<class
...
</class>

<class
...
</class>

</hibernate-mapping>
```

**One Statement per Line**

**Severity:** Medium

**Summary**

Each statement should be on its own line.

**Description**

This audit rule checks for statements that occur on the same line as a previous statement and flags them.

**Example**

The second assignment statement would be flagged as needing to be on its own line:

```
x = 0; y = 0;
```

**Overloaded Equals**

**Severity:** Medium

**Summary**

The equals method should always take a parameter of type Object.

**Description**

This audit rule looks for declarations of the method equals whose single parameter has a declared type different from java.lang.Object. Overloading the equals method can easily lead to situations where a.equals(b) != b.equals(a).

**Example**

```
public boolean equals(String string)
{
...
}
```

**Overloaded Methods**

**Severity:** Medium

**Summary**

Overloading method names can cause confusion and errors.

**Description**

This audit rule finds methods that are overloaded. Overloaded methods are methods that have the same name and the same number of parameters, but do not have the same types of parameters. Such methods can cause confusion and errors because it is not always obvious which method will be selected at run time.

**Example**

```
public void process(Person person)
public void process(Employee employee)
```

**Override both equals() and hashCode()**

**Severity:** Medium

**Summary**

Classes should override both equals() and hashCode() if they override either.

**Description**

This audit rule finds classes in which either the equals() or hashCode() method has been overridden, but not both.

**Example**

The following class declaration will be flagged as a violation because it overrides the method equals() but does not override the method hashCode():

```
public class Employee
{
private String name;

...
```

```
public boolean equals(Object object)
{
return object instanceof Employee
&& getName().equals(((Employee) object).getName());
}
}
```

**Override Clone Judiciously**

**Severity:** Medium

**Summary**

Be careful when defining clone. It is complex and not fully specified.

**Description**

This audit rule helps to ensure that clone is used properly. It checks the following items:
1. Either the class is final or the clone method calls super.clone().
2. Either the class is abstract or the clone method is declared public.
3. Either the class is final or the clone method declaration has the proper throws clause.

**Overriding a Non-abstract Method with an Abstract Method**

**Severity:** Medium

**Summary**

An abstract method should not override a non-abstract method.

**Description**

This audit rule finds abstract methods that override a non-abstract method. Such a case usually represents a violation of the inherited contract.

**Example**

Given the following class:

```
public class Widget
{
public int getPartCount()
{
```

```
return 0;
}
}
```

The method getPartCount() would be flagged as a violation in the following class:

```
public abstract class CompositeWidget extends Widget
{
public abstract int getPartCount();
}
```

**Overriding a Synchronized Method with a Non-synchronized Method**

**Severity:** Medium

**Summary**

A non-synchronized method should not override a synchronized method.

**Description**

This audit rule finds non-synchronized methods that override a synchronized method. Such a case usually indicates thread-unsafe code.

**Example**

```
public class Widget
{
public synchronized int getPartCount()
{
return 0;
}
}
```

The method getPartCount() would be flagged as a violation in the following class:

```
public class CompositeWidget extends Widget
{
public int getPartCount()
{
return children.size();
}
}
```

**Overriding Private Method**

**Severity:** Medium

**Summary**

Methods cannot override a private method.

**Description**

This audit rule looks for methods that look like they were intended to override a method from a superclass but fail because the inherited method is defined to be a private method.

**Example**

Given the following class:

```
public class TreeNode
{
private int getChildCount()
{
return children.size();
}
}
```

The method getChildCount() would be flagged as a violation in the following class:

```
public abstract class LeafNode extends TreeNode
{
private int getChildCount()
{
return 0;
}
}
```

**Package Javadoc**

**Severity:** Low

**Summary**

Packages should have Javadoc comments in a file named "package.html".

**Description**

This audit rule checks all of the packages to ensure that there is a file named "package.html" that can be used by the Javadoc program to create package-level Javadoc information.

**Package Naming Convention**

**Severity:** Medium

**Summary**

Package names should conform to the defined standard.

**Description**

This audit rule checks the names of all packages as defined by the package declarations at the beginning of each compilation unit.

**Example**

If the rule has been configured to only allow package names consisting of lower case letters, the package name "utilitiyClasses" would be flagged as a violation because it contains an upper case letter.

**Package Prefix Naming Convention**

**Severity:** Medium

**Summary**

Package names should begin with either a top-level domain name or a two-letter country code.

**Description**

This audit rule checks the names of all packages as defined by the package declarations at the beginning of each compilation unit to ensure that the first identifier is either a top-level domain name or a two-letter country code.

**Example**

For example, the following package declaration would be flagged as a violation because "freeware" is neither a top-level domain name nor a two-letter country code:

```
package freeware.model;
```

**Package Structure**

**Severity:** Medium

**Summary**

Packages should follow a prescribed structure.

**Description**

This audit rule looks for package names whose structure does not conform to the configured standard.

**Example**

If the rule were configured to require package names to start with a prefix of "org.apache" and have at least one segment representing the name of the project, with "ant", "jakarta", and "struts" on the list, the name "justforfun" in the following package name would be flagged as a violation:

```
package org.apache.justforfun.nonsense;
```

**Parenthesize Condition in Conditional Operator**

**Severity:** Low

**Summary**

The condition in a conditional operator should be parenthesized if it contains a binary operator.

**Description**

This audit rule looks for uses of the conditional operator in which the condition contains a binary operator but has not been parenthesized.

**Example**

The following expression would be flagged as a violation because the condition is not parenthesized:

```
childCount = children == null ? 0 : children.size();
```

**Parse Method Usage**

**Severity:** Low

**Summary**

The parseXXX() methods for Numerics should not be used in an internationalized environment.

**Description**

This audit rule checks for the use of the parseXXX() methods for Numerics in an internationalized environment. Numeric formats differ with region and language, so consistent results cannot be counted on.

**Example**

The following use of the parseInteger() method would be flagged as a violation:

```
myInt = Integer.parseInteger("12345");
```

**Password in File**

**Severity:** Medium

**Summary**

Storing passwords in a configuration file is a security risk.

**Description**

This audit rule violates all instances of the password retrieval from a property file.

**Security Implications**

Storing passwords in a configuration file can provide an attacker with the data that can be used to access a protected resource.

**Example**

The following code retrieves password from a file and is thus marked as a violation:

```
Properties props = new Properties();
props.load(new FileReader("app.properties"));
String password = props.getProperty("password");
```

**Path Manipulation**

**Severity:** High

**Summary**

User input might be getting used to create a path to a resource.

**Description**

Path Manipulation occurs when user input is used directly as a path or part of a path to a resource.

To detect violations, this audit rule searches the code for path creation statements such as `File file = new File(..)` and traces where the path could have come from. In cases where the source of the path is user input, such as data from a servlet request, `javax.servlet.ServletRequest.getParameter(java.lang.String)`, or from a SWT Text widget, `org.eclipse.swt.widgets.Text.getText()`, a violation is created.

These two sets of methods, the locations where tainted user data can come from and the methods used to create paths, are editable by the user. If methods are missing that are in a common package (such as `java.lang.*`), please let CodePro support know.

**Security Implications**

When a malicious user can enter a path directly, system or application files that are assumed protected from users can become vulnerable.

**Example**

The invocation of the constructor `File(..)` would be flagged as a violation since it uses the path name information passed from a servlet request:

```
ServletRequest servletRequest = ...;
String fileName = servletRequest.getParameter("fileName");
File file = new File("..." + fileName);
```

**Plain Text Password**

**Severity:** Medium

**Summary**

Storing passwords as plain text makes them easily accessible by an attacker.

**Description**

This audit rule violates all usages of an unencrypted password retrieved from a property file.

**Security Implications**

Storing passwords in a configuration file as plain text can provide a malicious user or an attacker with a password that can be later used to access secured data. Passwords should be stored in an encoded state and decoded using some strong encryption algorithm.

**Example**

The following code retrieves a password from a file and uses it directly to access a database without decoding. It is thus marked as a violation:

```
Properties props = new Properties();
props.load(new FileReader("security.properties"));
String username = props.getProperty("username");
String password = props.getProperty("password");
Connection c = source.getConnection(username, password);
```

**Platform Specific Line Separator**

**Severity:** High

**Summary**

Line separators should not be hard-coded because they are platform dependent.

**Description**

Line separators should not be hard-coded because they are platform dependent.

**Example**

The following use of the escape sequence "\n" would be flagged as a violation because not all platforms use it as a line separator:

```
text = title + "\n\n" + message;
```

**Pluralize Collection Names**

**Severity:** Medium

**Summary**

The names of collection-valued variables should be plural.

**Description**

This audit rule checks for collection-valued variables whose name is singular. A collection-valued variable is any variable whose type is either an array type or a subtype of the type java.util.Collection.

**Example**

The following field declaration would be flagged because its name is singular:

```
private List employee;
```

**Possible Null Pointer**

**Severity:** Medium

**Summary**

A pointer is being dereferenced when it might be null.

**Description**

This rule identifies places where an object-valued variable is being dereferenced without first ensuring that it cannot be null.

**Example**

Given the following method declaration:

```
public String[] split(String string)
{
int index = string.indexOf(":");
...
}
```

The invocation of the indexOf method would be flagged.

**Potential Infinite Loop**

**Severity:** Medium

**Summary**

Some loops can be written in such a way that they will never terminate. This is bad practice, and usually not intended.

**Description**

Loops can exit in several ways. Either their exit condition can be satisfied, an exception can be thrown, a value can be returned, or a `break` or a `continue` can transfer control out of the loop. In the body of the loop, something should happen to either modify the value of the exit condition, or modify the value of a condition leading to a return, throw, or branching statement.

**Example**

The following would be flagged as a violation, since the value of `a` is not changed in the body of the loop.

```
int a = 0; int b = 2;
while (a < 10) {
b++;
}
```

**Pre-compute Constant Calculations**

**Severity:** Medium

**Summary**

It is faster and more accurate to pre-compute the value of a `Math` operation involving only constants.

**Description**

This rule finds uses of static methods in the `Math` class which are passed constant values. It is faster, and often more accurate to pre-compute these values.

**Example**

```
double foo = Math.cos(0);
```

**Prefer Interfaces to Abstract Classes**

**Severity:** Low

**Summary**

It is better to define abstract types as interfaces than classes.

**Description**

This rule identifies types defined by abstract classes. It flags abstract classes that define public methods that are not defined in an interface implemented by the class.

Rather than defining an abstract class to create a type, define an interface instead. In addition to the interface, include an abstract class that provides a skeletal implementation of the type, which implements the interface.

**Prefer Interfaces To Reflection**

**Severity:** Low

**Summary**

Consider using interfaces instead of reflection.

**Description**

This audit rule identifies uses of reflective capabilities and flags them for review. Much of what is accomplished via reflection can be done, faster and simpler, with judicious use of interfaces.

**Preferred Expression**

**Severity:** Low

**Summary**

Some expressions are preferred over other equivalent expressions.

**Description**

This audit rule searches for expressions that can be replaced with other equivalent and more preferred expressions. Specifically Colors can be replaced with Color constants and Booleans with Boolean constants.

**Example**

The expression

```
new Color(0, 0, 0)
```

can be replaced by

```
Color.black
```

and the expression

```
new Boolean(true)
```

can be replaced by

```
Boolean.TRUE
```

**Prevent Overwriting Of Externalizable Objects**

**Severity:** Medium

**Summary**

Disallows Externalizable objects to be overwritten.

## Description

All classes that implement java.io.Externalizable must declare the public readExternal method.

This audit rule declares that every subclass of Externalizable must declare readExternal and that readExternal must match the following pattern:

```
public synchronized void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
if (! initialized) {
initialized = true;
*
} else {
throw new IllegalStateException();
}
}
```

The boolean named "initialized", may have any valid name, and can be accessed via the this keyword, but cannot be re-assigned anywhere else within the class as such a reassignment could easily invalidate the security measures being taken.

## Security Implications

When readExternal is not overridden in this way, malicious users can call readExternal after the class has already been initialized and potentially have the internal state of the class overwritten.

## Example

The following class would be flagged as readExternal is not declared as synchronized:

```
public class A implements Externalizable {

boolean initialized = false;
public void readExternal(ObjectInput in) {
if (! initialized) {
initialized = true;
// initialize the object } else {
throw new IllegalStateException();
}
}

public void writeExternal(ObjectOutput out) throws IOException {
*
}
}
```

## Process Control

**Severity:** High

**Summary**

Process controlling code might be receiving data from the user or other unsafe sources.

**Description**

Process Control occurs when the user is able to enter data directly into code that effects the process of the runtime. This can occur when the user has access to specify system commands or when then user has access to specify which libraries are loaded by the application.

To detect violations, this audit rule searches the code for process controlling code methods such as `java.lang.System.load(..)` and traces where the data could have come from. In cases where the source of the query is user input, such as data from a servlet request, `javax.servlet.ServletRequest.getParameter(java.lang.String)`, or from a SWT Text widget, `org.eclipse.swt.widgets.Text.getText()`, a violation is created.

These two sets of methods, the locations where tainted user data can come from and the methods used to query the database, are editable by the user.If methods are missing that are in a common package (such as `java.lang.*`), please let CodePro support know.

**Security Implications**

Successful Process Control attacks can potentially allow the user to specify source code to be executed.

**Example**

The invocation of the method `System.loadLibrary(..)` would be flagged as a violation since it uses the library name information passed from a SWT widget:

```
Text text = ...;
String library = text.getText();
System.loadLibrary(library);
```

**Proper Finalize Usage**

**Severity:** Medium

**Summary**

Don't call various methods from a finalize() method.

**Description**

Avoid calling various methods from a finalize() method. Specifically avoid calling any methods from the ClassLoader, Runnable or PortbaleRemoteObject classes. Also avoid calling any methods from the java.applet, java.beans, java.io, java.lang.ref, java.net, java.rmi, java.security, java.sql, javax.naming, javax.sql, javax.transaction or org.omg packages.

**Proper Servlet Usage**

**Severity:** Medium

**Summary**

Don't call various methods from a Servlet.

**Description**

Avoid calling various methods from a Servlet. Specifically avoid calling any methods from the Thread, ThreadGroup, System, Process, Runtime, DriveManager or Compiler classes. Also avoid calling the Runnable.run() method as well as any methods from the java.awt, javax.swing, java.lang.reflect, javax.sound, java.util.jar or java.util.zip packages.

**Property File Must Exist**

**Severity:** Medium

**Summary**

Property file referenced for the placeholder configurer bean must exists.

**Description**

If a bean uses one of the following classes:

```
org.springframework.beans.factory.config.PropertyPlaceholderConfigurer
org.springframework.beans.factory.config.PreferencesPlaceholderConfigurer
org.springframework.web.context.support.ServletContextPropertyPlaceholderConf
igurer
```

this audit rule checks for a 'location' property and the existence of the property file referenced in this property.

**Example**

The file name jdbc.properties would be flagged if the file jdbc.properties does not exist:

```
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer
">
<property name="location">
<value>jdbc.properties</value>
</property>
</bean>
```

**Protected Member in Final Class**

**Severity:** Medium

**Summary**

Members of a final class should not be protected.

**Description**

This audit rule finds members that have a visibility of protected but are defined in a final class. There is no reason for them to be given a visibility of protected because there cannot be any subclasses.

**Example**

The method getSalary() would be flagged as a violation in the following example:

```
public final class Employee
{
protected double getSalary()
{
...
}
}
```

**Public Constructor in Non-public Type**

**Severity:** Medium

**Summary**

Constructors in non-public classes should not be public.

**Description**

This audit rule finds public constructors in non-public classes. There is no value in providing a public constructor because a non-public class cannot be instantiated outside the package in which it is defined. Reduce the access of the constructor to match that of the class itself.

**Example**

The constructor in the following example would be flagged as a violation because the class only has package visibility:

```
class EmployeeData
{
public EmployeeData()
{
...
}
}
```

**Public Nested Type in Package Visible Type**

**Severity:** Medium

**Summary**

Types declared within package visible types should not be public.

**Description**

This audit rule looks for declarations of public nested types within a package visible type. Because the outer type limits the visibility of the nested type, there is no reason for it to be declared as public, and such a declaration can be misleading.

**Example**

The following nested type would be flagged as a violation:

```
class OuterClass {
public class InnerClass {
}
}
```

**Questionable Assignment**

**Severity:** Low

**Summary**

Questionable assignments may lead to semantic errors.

**Description**

This audit rule checks for assignments to for loop variables within the loop body or assignments to method parameters within the body of the method. While these may sometimes prove useful they often have unintended side effects. Be sure you really mean to make these assignments!

**Example**

The assignment to "index" in the following method would be flagged as a violation because it is a method parameter:

```
public int lastIndexOf(Object[] array, Object value, int index)
{
index = array.size() - 1;
while (index >= 0) {
...
}
}
```

**Questionable Name**

**Severity:** High

**Summary**

Questionable names may indicate sloppy code.

**Description**

This audit rule checks for questionable names of variables, fields, methods, and types. A questionable name may indicate sloppy code.

Three types of questionable names are flagged:

1. Names that occur in a user-configurable list.

2. Names that are shorter than a user-configurable size. Short names can be useful as for loop variables or in catch clauses so short names are not flagged if they occur in those two places.

3. Names that are longer than a user-configurable size.

Names that you always want to be accepted can be added to a list at the bottom of the preference pane. These names will be accepted whether or not they break any of the 3 rules given above.

**Example**

The variable "c" would be flagged as a violation in the following method because it is too short, but the variable "i" would not be because it is used as a loop variable:

```
public int getNonZeroElementCount()
{
int c;

c = 0;
for (int i = 0; i < elements.size(); i++) {
...
}
return c;
}
```

**Realm Debug Enabled**

**Severity:** Medium

**Summary**

In Tomcat 4.x and earlier versions JAAS Realm debug level of 3+ logs users' credentials.

**Description**

Tomcat realms log some information, its amount depending on the debug level regulated by the `debug` attribute. This audit rule violates declarations of `debug` attribute of values of 3 and higher for the JAAS `<Realm>` declarations in Tomcat server configurations.

**Security Implications**

For most realms, debug level of 2 and higher logs the username, but JAASRealm does also log the password when debug level is 3 or higher. Having the password logged in plain text is a security risk that should be avoided, attacker could access the log file and retrieve passwords.

**Example**

The following code declares a JAASRealm with a debug level of 3 and will be marked as a violation:

```
<Realm className="org.apache.catalina.realm.JAASRealm" appName="MyFooRealm"
userClassNames="org.foobar.realm.FooUser"
roleClassNames="org.foobar.realm.FooRole" debug="99">
```

**Recursive Call With No Check**

**Severity:** Medium

**Summary**

Don't recursively call a method without a conditional check.

**Description**

This audit rule flags methods that recursively call themselves with no conditional check, or return escape. Violations are either infinite loops, or the logic of the method relies on exceptions being thrown. In the first case the infinite loop needs to be removed. In the second case, thrown exceptions should not be relied on as they are much more expensive than writing the equivalent conditional.

**Example**

```
private void countDownToZero(int i) {
System.out.print("i = " + i);
i--;
countDownToZero(i);
// never reached!
if(i == 0) {
return;
}
}
```

**Redirected With Password**

**Severity:** Medium

**Summary**

Passwords should not be passed in redirect URLs.

**Description**

This audit rule violates usage of a password in the construction of a HTTP GET request URL.

**Security Implications**

Sending redirect to the browser will result in a HTTP GET request. Because the GET request is not considered to be security-sensitive data, it will be easily accessible - it could be sniffed, logged by a proxy, or by an HTTP server itself. This adds more places that could be later used by an attacker to retrieve the password and the data it protects.

**Example**

The following code sends a password in the redirect and will be marked as a violation.

```
resp.sendRedirect(req.getRequestURI() + "&myPassword=" + myPassword);
```

**Redundant Assignment**

**Severity:** High

**Summary**

Redundant assignments should never be used.

**Description**

This audit rule checks for the assignment of a variable to itself. This often indicates a missing qualifier, such as "this." for one or the other of the identifiers.

**Example**

```
public void setName(String name)
{
name = name;
}
```

**Referenced Class Not Defined**

**Severity:** Medium

**Summary**

Classes referenced in the configuration file (*.hbm.xml) should be declared.

**Description**

This audit rule checks for references within a Hibernate configuration file to classes that are not declared in the associated project. Hibernate will not work correctly if undeclared classes are referenced. Either the references should be removed or the classes should be declared.

**Example**

In the following entry (from a *.hbm.xml file), the value of the "name" attribute would be flagged

```
<class
name="class.not.exists" ... />
```

**Referenced Class Not Defined**

**Severity:** Medium

**Summary**

Classes referenced in Spring configuration files should be declared.

**Description**

This audit rule checks for references within a configuration file to classes that are not declared in the associated project. Either the references should be removed or the classes should be declared.

**Example**

In the following entry (from a XML configuration file), the value of the "class" attribute would be flagged:

```
<bean
id="test"
class="class.not.exists/">
```

**Reflection Injection**

**Severity:** High

**Summary**

Request parameters and other tainted data should not be passed into methods accessing classes, methods or fields via reflection without sanitizing.

**Description**

This rule violates usage of an unvalidated user input as a part of class or method names referred to by reflection calls.

**Security Implications**

Only trusted data should be used in reflection class, otherwise an attacker can instantiate a class or invoke a method of their choice, potentially bypassing the security system of an application. This can lead to faulty or unexpected behaviour.

**Example**

The following code selects a class to instantiate with the use of unsanitized data retrieved from `HttpServletRequest` parameter and does not clean this data before instantiating the class:

```
LdapContext ctx = getDirContext();
String userName = req.getParameter("user");
String filter = "(user=" + userName + ")";
try {
NamingEnumeration users = ctx.search("ou=People,dc=example,dc=com", filter,
null);
```

**Reflection Method Usage**

**Severity:** Low

**Summary**

Don't use specific reflection methods.

**Description**

Don't use Class getMethod(), getField(), getDeclaredMethod() or getDeclaredField() methods in production code.

**Relative Access to Enterprise Schemas**

**Severity:** Medium

**Summary**

Enterprise schemas must be pulled into your project and the link to the enterprise schemas must be relative.

**Description**

This audit rule looks for places where the access to enterprise schemas is made using an absolute URI.

**Example**

The following schema location specification would be flagged as a violation because it uses an absolute URI:

```
schemaLocation="http://Enterprise_SalesPreferenceService.xsd"
```

It should be replaced by something like the following:

```
schemaLocation="Enterprise_Quote.xsd"
```

**Relative Library Path**

**Severity:** Medium

**Summary**

Always use absolute paths when loading libraries.

**Description**

This audit rule looks for places where libraries are loaded using a relative file path.

**Security Implications**

Loading libraries without specifying an absolute path can cause the program to load malicious libraries supplied by an attacker.

**Example**

The following code uses `System.loadLibrary()` to load code from a native library named `library.dll`, which is normally found in a standard system directory.

```
System.loadLibrary("library.dll");
```

**Repeated Assignment**

**Severity:** High

**Summary**

A single variable should not be assigned the same value multiple times.

**Description**

This audit rule checks for multiple assignments of the same value to a single variable within the same statement.

**Example**

The following assignment would be flagged because the variable x is assigned the same value twice:

```
x = y = x = 0;
```

**Replace Synchronized Classes**

**Severity:** Medium

**Summary**

Synchronized classes should only be used if the synchronization is necessary.

**Description**

This audit rule flags uses of synchronized classes which could be replaced with faster non-synchronized replacements.

**Example**

The following would be flagged because HashMap could be used instead:

```
Hashtable foo = new Hashtable();
```

**Request Message Naming Convention**

**Severity:** Medium

**Summary**

Request messages must have the same name as the operation.

**Description**

This audit rule looks for message request declarations that do not explicitly specify its name as the same as the operation's name.

**Example**

The following operation declaration does not specify its request name as the same as the operation's name and would thus be marked as a violation:

```
<wsdl:definitions .... >
<wsdl:portType .... >
<wsdl:operation name="nmtoken" parameterOrder="nmtokens">
<wsdl:input message="qname"/>
<wsdl:output name="nmtoken" message="qname"/>
<wsdl:fault name="nmtoken" message="qname"/>
</wsdl:operation>
</wsdl:portType>
</wsdl:definitions>
```

**Request Parameters In Session**

**Severity:** High

**Summary**

Request parameters and other tainted data should not be passed into Session without sanitizing.

**Description**

Sessions should only store trusted data, so that the developer accessing the data stored in a session would not have to decide whether to sanitize it.

**Security Implications**

Data stored in a session is usually considered by a developer as a safe one to use. If this data is not checked, it could get into the security-sensitive parts of an application, opening it to all kinds of injection attacks.

**Example**

The following code uses receives data via `HttpServletRequest#getParameter()` call and does not clean it before putting it into the session:

```
login = request.getParameter("login");
session.setAttribute(ATTR_LOGIN, login);
```

**Resource Injection**

**Severity:** High

**Summary**

Users input should not be able to specify resource identifiers.

**Description**

Resource Injection occurs when the user is able to provide resource identifiers such as a port number.

To detect violations, this audit rule searches the code for ServerSocket creation methods such as `java.net.ServerSocket.ServerSocket(int)` and traces where the query data could have come from. In cases where the source of the port number is user input, such as data from a servlet request, `javax.servlet.ServletRequest.getParameter(java.lang.String)`, or from a SWT Text widget, `org.eclipse.swt.widgets.Text.getText()`, a violation is created.

These two sets of methods, the locations where tainted user data can come from and the methods used to identify resources, are editable by the user.If methods are missing that are in a common package (such as `java.lang.*`), please let CodePro support know.

**Security Implications**

Successful Resource Injection attacks can potentially allow malicious users to gain access to resources not intended by the application.

**Example**

The invocation of the constructor `ServletRequest(..)` would be flagged as a violation since it uses the port number passed from a servlet request:

```
ServletRequest servletRequest = ..;
String portNumberStr = servletRequest.getParameter("portNumber");
int portNumber = (new Integer(portNumberStr)).intValue();
ServerSocket serverSocket = new ServerSocket(portNumber);
```

**Resource URL Manipulation**

**Severity:** High

**Summary**

Request parameters and other tainted data should not be used as a part of a URL accessed via `getResource()` or `getResourceAsStream()` calls.

**Description**

Only trusted data should be used when creating a URL to access.

**Security Implications**

Otherwise an attacker can specify an arbitrary path, potentially accessing a filesystem or other sensitive data.

**Example**

The following code uses data from an HTTP request parameter to access a resource and does not validate this data:

```
String resURL = req.getParameter("resURL");
return ResourceRetriever.class.getResource(resURL);
```

**Response Message Naming Convention**

**Severity:** Medium

**Summary**

Input and output of operations must be wrapped with a message request and response. Response messages must be named as follows: `<OperationName>Response`.

**Description**

This audit rule looks for message response declarations that specify a name that does not match the pattern `<OperationName>Response`.

**Example**

The following operation declaration does not specify its response name as equal to `<OperationName>Response` and would thus be marked as a violation:

```
<wsdl:definitions .... >
<wsdl:portType .... >
<wsdl:operation name="nmtoken" parameterOrder="nmtokens">
<wsdl:input message="qname"/>
<wsdl:output name="nmtoken" message="qname"/>
<wsdl:fault name="nmtoken" message="qname"/>
</wsdl:operation>
</wsdl:portType>
</wsdl:definitions>
```

**Restricted Packages**

**Severity:** High

**Summary**

Some packages contain internal implementation details and should never be imported.

**Description**

This audit rule allows the user to specify a list of package names that should not be imported and will check that none of the packages on the list are imported. For example, if the package name internal is included in the list, then any import that contains the identifier internal as one of the components of the package name would be flagged.

**Example**

```
import org.eclipse.core.internal.runtime.*;
```

**Restricted Superclasses**

**Severity:** Medium

**Summary**

Some classes should never be directly subclassed even though they cannot be marked as final.

**Description**

This audit rule finds classes that directly subclass any of a list of classes that should never be directly subclassed. It is typically the case that there are subclasses of these classes that can be subclassed.

**Example**

If the rule is configured to disallow direct subclasses of java.lang.Throwable, the following class declaration would be flagged as a violation:

```
public class QuestionableResult extends Throwable
{
...
}
```

**Rethrown Exceptions**

**Severity:** Medium

**Summary**

Some exceptions should be rethrown.

**Description**

This audit rule finds catch clauses that do not rethrow the original exception. The list initially includes Error and ThreadDeath. For example, if ThreadDeath is not rethrown, then the thread will not terminate.

**Example**

If the rule has been configured to require that instances of the class java.lang.ThreadDeath must be rethrown, the following catch clause would be flagged as a violation:

```
try {
...
} catch (ThreadDeath exception) {
resource.release();
}
```

**Return Boolean Expression Value**

**Severity:** Medium

**Summary**

Return the value of a boolean expression directly.

**Description**

Rather than testing a boolean value in an if-statement then returning true or false, simply return the value of the boolean expression directly.

**Example**

```
if (booleanValue)
return true;
else
return false;
```

Should be changed to:

```
return booleanValue;
```

**Return Constant From getComponentType**

**Severity:** Medium

**Summary**

The method getComponentType() is expected to return a constant string.

**Description**

This audit rule looks for implementations of getComponentType() that return a value other than a constant string (i.e. a literal or a final static field). Because the value returned must match the one in the XML file, there is no point in doing any calculations to produce it. Doing so reduces readability of the source and also prevents automatic checking of the return value.

**Example**

The following return statement will be flagged as a violation.

```
public class RoundButtonTag extends UIComponentTag {
...
public String getComponentType() {
return "CheatyButtonNr" + System.currentTimeMillis();
}
...
}
```

**Return Constant From getFamily**

**Severity:** Medium

**Summary**

The method getFamily() is expected to return a constant string.

**Description**

This audit rule looks for implementations of getFamily() that return a value other than a constant string (i.e. a literal or a final static field). Because the value returned must match the one in the XML file, there is no point in doing any calculations to produce it. Doing so reduces readability of the source and also prevents automatic checking of the return value.

**Example**

The following return statement will be flagged as a violation.

```
public class RoundButton extends UIComponentTag {
...
public String getFamily() {
return "RoundButtonNr" + System.currentTimeMillis();
}
...
}
```

**Return Constant From getRendererType**

**Severity:** Medium

**Summary**

The method getRendererType() is expected to return a constant string.

**Description**

This audit rule looks for implementations of getRendererType() that return a value other than a constant string (i.e. a literal or a final static field). Because the value returned must match the one in the XML file, there is no point in doing any calculations to produce it. Doing so reduces readability of the source and also prevents automatic checking of the return value.

**Example**

The following return statement will be flagged as a violation.

```
public class RoundButtonTag extends UIComponentTag {
...
public String getRendererType() {
return "RoundButtonNr" + System.currentTimeMillis();
}
...
}
```

**Return in Finally**

**Severity:** Medium

**Summary**

Finally blocks should not contain a return statement.

**Description**

This audit rule finds places where a return statement is contained in a finally block.

**Example**

The following return statement would be flagged as a violation:

```
try {
...
} finally {
return 0;
}
return array.length;
```

**Reusable Immutables**

**Severity:** Medium

**Summary**

Objects that cannot be modified at run-time should be created as static constants.

**Description**

This audit rule finds the creation of some kinds of immutable objects. An immutable object is an object whose state cannot be modified at run-time. Such objects should be created as constants (static final fields) in order to reduce the amount of garbage that must be collected.

**Example**

```
new Integer(5);
new int[0];
```

**Reuse DataSources for JDBC Connections**

**Severity:** Medium

**Summary**

Using method-local DataSource could lead to a concurrent access to the database.

**Description**

This rule looks for the creation of `DataSource` objects in methods.

**Security Implications**

Using method-local DataSource objects could lead to an unprotected concurrent access to the database, which could result in an unexpected state of either the application or its data.

**Example**

The following method creates a `DataSource` object and would thus be marked as violation:

```
public void local() {
....
DataSource ds = null;
....
}
```

**Rollback Transaction on Exception**

**Severity:** High

**Summary**

Rollback transactions if an exception occurs.

**Description**

If the session throws an exception (including any SQLException), you should immediately rollback the transaction, call Session.close() and discard the session instance because certain methods of Session will not leave the session in a consistent state. This audit rule looks for places where the transaction will not be rolled back correctly.

**Example**

The following method would be flagged because it doesn't rollback the transaction if an exception occurs.

```
private void transactionExample(...) {
...
Session session = factory.openSession();
Transaction tx = session.beginTransaction();
// do some work
...
tx.commit();
...
}
```

It should be replaced by something like the following

```
private void transactionExample(...) {
...
Session session = factory.openSession();
Transaction tx = null;
try {
tx = session.beginTransaction();
// do some work
...
tx.commit();
} catch (Exception e) {
if (tx != null) tx.rollback();
throw e;
} finally {
session.close();
}
...
}
```

**Runtime Method Usage**

**Severity:** Low

**Summary**

Don't use specific Runtime methods.

**Description**

Don't use Runtime gc(), runFinalization() or runFinalizersOnExit() methods in production code.

**Same Validator Name**

**Severity:** Medium

**Summary**

Duplicating validator names cause errors in validation process.

**Description**

This audit rule violates declarations of a validators with the same `name` attribute.

**Security Implications**

From two validators with the same name only one will be executed. Duplicating validator names is usually a sign that validation in the application is out of order and should be fixed. Failing in doing so will allow the attacker to inject some kind of invalid data into application.

**Example**

The following `validators.xml` configuration contains two declared validators with the same name; these declarations would be violated:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator Config
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-config-1.0.dtd">
<validators>
<validator name="custom" class="org.some.validation.old.CustomValidator1"/>
<validator name="custom" class="org.some.validation.new.CustomValidator2"/>
</validators>
```

**Serializable Usage**

**Severity:** Medium

**Summary**

Serializable classes should be define properly.

**Description**

This audit rule finds classes that are defined as Serializable (that implement the interface java.io.Serializable) that have specific problems or that do not need to be defined as Serializable.

**Example**

The following class would be flagged as a violation because it does not define any instance fields:

```
public class Utilities implements Serializable
{
public static double computeInterest(double amount, double interestRate,
double payment)
{
...
}
}
```

**Serializeability Security**

**Severity:** Medium

**Summary**

When an Object implements Serializeable, adversaries can sometimes serialize the Object into a byte array, because of this, secure classes must be conscious of Serializeable.

**Description**

This audit rule enforces the implementation of the following in all classes that implement java.io.Serializeable:

```
private final void writeObject(ObjectOutputStream in) throws IOException {
new java.io.IOException("Class cannot be serialized");
}
```

This rule flags instances of Serializable that do not contain the following method signature:

```
private void writeObject(ObjectOutputStream in) throws IOException;
```

**Security Implications**

This audit rule aims to prevent adversaries from accessing the state of an Object by extending classes and retrieving the Object data by calling writeObject.

**Example**

The following method declaration would be flagged as a violation because it does not have the required parameter:

```
private void writeObject() throws IOException;
```

**Server Request Not Encrypted**

**Severity:** Medium

**Summary**

Websphere web service should only accept encrypted messages.

**Description**

This audit rule looks for declarations of Websphere web service provider extensions and violates the declarations which do not declare message encryption.

**Security Implications**

Message-level encryption is especially important for messages that are passed via such easily readable protocols as HTTP or SMTP, as well as the others. Encrypted message maintains the required level of security not depending on the protocol.

**Example**

The following code specifies the encryption constraints that messages consumed by a service provider must meet. If no encryption constraints, declared via `<requiredConfidentiality>` property, are defined, the whole `<securityRequestReceiverServiceConfig>` declaration is marked as violation:

```
<securityRequestReceiverServiceConfig>
<requiredConfidentiality>
<confidentialParts part="bodycontent"/>
<confidentialParts part="usernametoken"/>
</requiredConfidentiality>
...
</securityRequestReceiverServiceConfig>
```

**Server Request Not Signed**

**Severity:** Medium

**Summary**

Websphere web service should only accept signed messages.

**Description**

This audit rule looks for declarations of Websphere web service provider extensions and violates the declarations which do not declare signing the message.

**Security Implications**

Signing the message is important because this is the only way to provide the complete guarantee for the receiver of a message that the message was endeed sent by a specific sender. Otherwise a message could be intercepted and its contents could be changed.

**Example**

The following code verifies that that messages consumed by a service provider are signed. If they are not verified to be signed, as declared via `<requiredIntegrity>` property, the whole `<securityRequestReceiverServiceConfig>` declaration is marked as violation:

```
<securityRequestReceiverServiceConfig>
<requiredIntegrity>
<references part="body"/>
<references part="timestamp"/>
</requiredIntegrity>
...
</securityRequestReceiverServiceConfig>
```

**Server Request Timestamp Not Signed**

**Severity:** Medium

**Summary**

Websphere web service timestamps should be signed.

**Description**

This audit rule looks for declarations of Websphere web service provider extensions and violates the declarations which do not declare signing of timestamps.

**Security Implications**

Unsigned timestamp can be intercepted and replaced by an attacker, thus easing a task of forging the message.

**Example**

The following code declares a `<addReceivedTimestamp>` without mentioning it in a `<requiredIntegrity>` list of signed components; it would thus be marked as violation:

```
<securityRequestReceiverServiceConfig>
<requiredIntegrity>
<references part="body"/>
</requiredIntegrity>
<addReceivedTimestamp flag="true"/>
...
</securityRequestReceiverServiceConfig>
```

**Server Response Not Encrypted**

**Severity:** Medium

**Summary**

Websphere web service should only produce encrypted messages.

**Description**

This audit rule looks for declarations of Websphere web service provider extensions and violates the declarations which do not declare message encryption.

**Security Implications**

Message-level encryption is especially important for messages that are passed via such easily readable protocols as HTTP or SMTP, as well as the others. Encrypted message maintains the required level of security not depending on the protocol.

**Example**

The following code specifies the encryption constraints that messages produced by a service provider must meet. If no encryption constraints, declared via `<confidentiality>` property, are defined, the whole `<securityResponseSenderServiceConfig>` declaration is marked as violation:

```
<securityResponseSenderServiceConfig>
<confidentiality>
<confidentialParts part="bodycontent"/>
</confidentiality>
...
</securityResponseSenderServiceConfig>
```

**Server Response Not Signed**

**Severity:** Medium

**Summary**

Websphere web service should only produce signed messages.

**Description**

This audit rule looks for declarations of Websphere web service provider extensions and violates the declarations which do not declare signing the message.

**Security Implications**

Signing the message is important because this is the only way to provide the complete guarantee for the receiver of a message that the message was endeed sent by a specific sender. Otherwise a message could be intercepted and its contents could be changed.

**Example**

The following code verifies that that messages consumed by a service provider are signed. If they are not verified to be signed, as declared via `<integrity>` property, the whole `<securityResponseSenderServiceConfig>` declaration is marked as violation:

```
<securityResponseSenderServiceConfig>
<integrity>
<references part="body"/>
<references part="timestamp"/>
</integrity>
...
</securityResponseSenderServiceConfig>
```

**Server Response Timestamp Not Signed**

**Severity:** Medium

**Summary**

Websphere web service timestamps should be signed.

**Description**

This audit rule looks for declarations of Websphere web service provider extensions and violates the declarations which do not declare signing of timestamps.

**Security Implications**

Unsigned timestamp can be intercepted and replaced by an attacker, thus easing a task of forging the message.

**Example**

The following code declares a `<addCreatedTimestamp>` without mentioning it in a `<integrity>` list of signed components; it would thus be marked as violation:

```
<securityResponseSenderServiceConfig>
<integrity>
<references part="body"/>
</integrity>
<addCreatedTimestamp flag="true"/>
...
</securityResponseSenderServiceConfig>
```

**Server Timestamp Does Not Expire**

**Severity:** Medium

**Summary**

Websphere web service timestamps should have an expiration date.

**Description**

This audit rule looks for declarations of Websphere web service provider extensions and violates the declarations which do not declare expiration date for created timestamps.

**Security Implications**

Timestamp without an expiration date is valid indefinetely, thus allowing the attacker to delay and modify messages, perform replay attacks, etc.

**Example**

The following code declares a `<addCreatedTimestamp>` without an expiration date set through a `expires` property; it would thus be marked as violation:

```
<securityResponseSenderServiceConfig>
<integrity>
<references part="body"/>
<references part="timestamp"/>
</integrity>
<addCreatedTimestamp flag="true"/>
...
</securityResponseSenderServiceConfig>
```

**Server Uses Username Token**

**Severity:** Medium

**Summary**

UsernameToken authentication is insecure and should not be used.

**Description**

This audit rule looks for declarations of Websphere web service server extensions and violates the declarations which use UsernameToken for authentication.

**Security Implications**

A UsernameToken authentication could use an unencrypted password that could be intercepted by an attacker.

**Example**

The following code declares a `<requiredSecurityToken>` with a UsernameToken as an authentication token and would thus be marked as violation:

```
<securityRequestConsumerServiceConfig>
<requiredSecurityToken name="Token_123" uri="" localName="http://docs.oasis-
```

```
open.org/wss/2004/01/oasis-200401-wss-username-token-profile-
1.0#UsernameToken" usage="Required"/>
...
</securityRequestConsumerServiceConfig>
```

**Service Naming Convention**

**Severity:** Medium

**Summary**

WSDL service names should conform to a naming standard.

**Description**

1. The word 'Service' must be added as a suffix to the service name
to distinguish between services and components.
For example an AccountService vs. an AccountComponent.
2. Service name should not include any service type information within the name.
This includes Entity, Application, Business, Utility, Integration, and Wrapper.
3. Service name should be written in camel case beginning with an upper case character.

**Example**

The following service declaration does not end with 'Service' and would thus be marked as a violation:

```
<wsdl:service name="NMToken">
....
</wsdl:service>
```

**Service Operation Naming Convention**

**Severity:** Medium

**Summary**

WSDL service operation names should conform to a naming standard.

**Description**

Service Operations names must follow the format: xxxxxYyyyy, where:

-xxxxx is a verb indicating the operation.
-Yyyyy is the subject of the operation. For application services
this is often the service name.

Standard Service Operation Names:

These names must be used within operation names matching the specified description.
* retrieve - returns one or more instance of a business entity.
* update - updates the information for one or more instances
of a business entity.
* delete - deletes one or more entities from the system.
* create - creates one or more new entity instances in the system.
* search - returns summary of data based on some search criteria.

**Example**

The following service operation declaration has a name incompatible with the declared standards and would thus be marked as a violation:

```
<wsdl:operation name="nmtoken">
....
</wsdl:operation>
```

**Service Provider Request Security Not Configured**

**Severity:** Medium

**Summary**

Websphere service provider should use WS-Security extension in order to provide encryption needed to protect sensitive data from sniffing or other ways of man-in-the-middle attacks.

**Description**

This audit rule violates Websphere server deployment descriptor extension file (`ibm-webservices-ext.xmi`) that does not use <securityRequestConsumerServiceConfig> to declare the usage of WS-Security.

**Security Implications**

WS-Security extension should be used to protected messages that are passed through third-party services from man-in-the-middle attacks via proper encryption and authentication of data.

**Example**

The following `ibm-webservices-ext.xmi` configuration does not use <securityRequestConsumerServiceConfig> and would thus be violated:

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsext:WsExtension xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.etools.webservice.wsext="http://www.ibm.com/websphere/appserver
/schemas/5.0.2/wsext.xmi" xmi:id="WsExtension_1118244521562">
<wsDescExt xmi:id="WsDescExt_1119553210812" wsDescNameLink="RRDService">
<pcBinding xmi:id="PcBinding_1119553210812" pcNameLink="RRDService"/>
<pcBinding xmi:id="PcBinding_1118244521562" pcNameLink="RRDServiceCustom">
<serverServiceConfig xmi:id="ServerServiceConfig_1123269103509"/>
</pcBinding>
</wsDescExt>
</com.ibm.etools.webservice.wsext:WsExtension>
```

**Service Provider Response Security Not Configured**

**Severity:** Medium

**Summary**

Websphere service provider should use WS-Security extension in order to provide encryption needed to protect sensitive data from sniffing or other ways of man-in-the-middle attacks.

**Description**

This audit rule violates Websphere server deployment descriptor extension file (`ibm-webservices-ext.xmi`) that does not use <securityResponseGeneratorServiceConfig> to declare the usage of WS-Security.

**Security Implications**

WS-Security extension should be used to protected messages that are passed through third-party services from man-in-the-middle attacks via proper encryption and authentication of data.

**Example**

The following `ibm-webservices-ext.xmi` configuration does not use <securityResponseGeneratorServiceConfig> and would thus be violated:

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsext:WsExtension xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.etools.webservice.wsext="http://www.ibm.com/websphere/appserver
/schemas/5.0.2/wsext.xmi" xmi:id="WsExtension_1118244521562">
<wsDescExt xmi:id="WsDescExt_1119553210812" wsDescNameLink="RRDService">
<pcBinding xmi:id="PcBinding_1119553210812" pcNameLink="RRDService"/>
<pcBinding xmi:id="PcBinding_1118244521562" pcNameLink="RRDServiceCustom">
<serverServiceConfig xmi:id="ServerServiceConfig_1123269103509"/>
</pcBinding>
</wsDescExt>
</com.ibm.etools.webservice.wsext:WsExtension>
```

**Service Requester Request Security Not Configured**

**Severity:** Medium

**Summary**

Websphere service requester should use WS-Security extension in order to provide encryption needed to protect sensitive data from sniffing or other ways of man-in-the-middle attacks.

**Description**

This audit rule violates Websphere server deployment descriptor extension file (`ibm-webservicesclient-ext.xmi`) that does not use <securityRequestGeneratorServiceConfig> to declare the usage of WS-Security.

**Security Implications**

WS-Security extension should be used to protected messages that are passed through third-party services from man-in-the-middle attacks via proper encryption and authentication of data.

**Example**

The following `ibm-webservicesclient-ext.xmi` configuration does not use <securityRequestGeneratorServiceConfig> and would thus be violated:

```
<clientServiceConfig>
<securityResponseConsumerServiceConfig />
</clientServiceConfig>
```

**Service Requester Response Security Not Configured**

**Severity:** Medium

**Summary**

Websphere service requester should use WS-Security extension in order to provide encryption needed to protect sensitive data from sniffing or other ways of man-in-the-middle attacks.

**Description**

This audit rule violates Websphere server deployment descriptor extension file (`ibm-webservicesclient-ext.xmi`) that does not use <securityResponseConsumerServiceConfig> to declare the usage of WS-Security.

**Security Implications**

WS-Security extension should be used to protected messages that are passed through third-party services from man-in-the-middle attacks via proper encryption and authentication of data.

**Example**

The following `ibm-webservicesclient-ext.xmi` configuration does not use <securityResponseConsumerServiceConfig> and would thus be violated. `xmi:id` attrubute entries were removed for improving readability of the example, but they should be present in the real-life code for the configuration to work:

```
<clientServiceConfig>
<securityRequestGeneratorServiceConfig>
<securityToken name="RRDToken"
uri="http://www.ibm.com/websphere/appserver/tokentype/5.0.2"
localName="LTPA"/>
</securityRequestGeneratorServiceConfig>
</clientServiceConfig>
```

**Session Beans**

**Severity:** Medium

**Summary**

Session Beans should be properly defined.

**Description**

This audit rule finds problems with Session Beans:
- Session bean should be declared public
- Session bean should not be declared as abstract or final
- Session bean name should end with 'Bean'
- Session bean not implement a finalize() method
- ejbCreate() method should be declared public
- ejbCreate() method should not be declared as static or final
- ejbCreate() method should be declared void
- Session bean should implement at least one ejbCreate() method
- Session bean should implement a no-argument constructor

**Session Must Be Synchronized**

**Severity:** Medium

**Summary**

References to the session must be synchronized.

**Description**

This audit rule looks for references to the session object that are not synchronized on the session. Because the controller servlet creates only one instance of your Action class and uses this one instance to service all requests, you need to write thread-safe Action classes. In particular, all references to the session object should be synchronized.

**Example**

The following method would be flagged because of the references to session:

```
public ActionForward execute(ActionMapping mapping, ActionForm inForm,
HttpServletRequest request, HttpServletResponse response) throws Exception {
HttpSession session = request.getSession();
if (session != null && session.getAttribute("user") != null) {
session.setAttribute("user", (String) request.getParameter("username"));
return mapping.findForward("success");
}
return mapping.findForward("fail");
}
```

It should be replaced by something similar to the following:

```
public ActionForward execute(ActionMapping mapping, ActionForm inForm,
HttpServletRequest request, HttpServletResponse response) throws Exception {
```

```
HttpSession session = request.getSession();
synchronized (session) {
if (session != null && session.getAttribute("user") != null) {
session.setAttribute("user", (String) request.getParameter("username"));
return mapping.findForward("success");
}
return mapping.findForward("fail");
}
}
```

**Session-scope Form Needs reset()**

**Severity:** Medium

**Summary**

The reset() method in an ActionForm should be implemented if the form has session scope (it's not used for request scope).

**Description**

This audit rule looks for classes that implement a form bean used by a session-scoped action and checks ro ensure that those classes implement a reset method.

**Example**

If the configuration file contains following entries:

```
<action path="/viewsignin"
type="com.instantiations.struts.example.actions.LoginAction"
scope="session"
name="loginForm"
validate="false"
input="/index.jsp"
parameter="/signin.jsp">
```

and

```
<form-bean name="loginForm"
type="com.instantiations.struts.example.actionforms.LoginForm"/>
```

then the class LoginForm would be flagged if it does not define a reset method.

**Simple Type Element Naming Convention**

**Severity:** Medium

**Summary**

Use lower camel case when naming elements that are simple types.

**Description**

This audit rule looks for declarations of simple type names that are not using lower camel case.

**Example**

The following element declaration has simple type but its name is declared in upper camel case and would thus be marked as a violation:

```
<element name="FirstName" type="string" />
```

**Simple Type Naming Convention**

**Severity:** Medium

**Summary**

Use lower camel case when naming simple types.

**Description**

This audit rule looks for declarations of simple type names that are not upper lower camel case.

**Example**

The following simple type declaration has its name in lower camel case and would thus be marked as a violation:

```
<simpleType name="MySimpleTypeName" />
```

**Sleep method invoked in synchronized code**

**Severity:** Medium

**Summary**

Do not invoke the sleep method while a lock is held.

**Description**

This rule looks for places where the code invokes the method `sleep(long)` while a lock is held.

**Security Implications**

An invocation of the sleep method while a lock is being held could lead to very poor performance or deadlock. Use the wait method instead of the sleep method.

**Example**

The following invocation of the `sleep()` method will be flagged as a violation because a lock is being held.

```
public synchronized void myMethod(Object obj)
{
while(true){
Thread.sleep(1000);
}
}
```

**Sockets in Servlets**

**Severity:** Medium

**Summary**

Sockets should almost never be used in a servlet environment as manual implementations of data transfer protocols compromises security.

**Description**

According to the J2EE Platform Specification, the only allowed usage of low-level socket interfaces is to establish connections to legacy systems.

**Security Implications**

Implementing protocols from scratch is error-prone and can easily compromise the security of the application.

**Example**

The following code uses sockets to establish a connection to the other system and would thus be marked as violation:

```
public class ProxyRequestServlet extends HttpServlet {
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
Socket s = new Socket("localhost", 8080);
}
}
```

**Source Length**

**Severity:** Low

**Summary**

Methods, constructors and initializers should be limited in length.

**Description**

This audit rule checks for methods, constructors and initializers that are more that the specified number of lines in length.

**Space After Casts**

**Severity:** Low

**Summary**

Casts should be followed by white space.

**Description**

This audit rule checks for uses of type casts to ensure that there is at least one space after the type.

**Example**

The following cast expression would be flagged as a violation because there is no space between the type and the expression:

```
value = (String)table.get(key);
```

**Space After Commas**

**Severity:** Low

**Summary**

Commas should be followed by white space.

**Description**

This audit rule checks for uses of commas to ensure that there is a space after the comma, but no space before the comma.

**Example**

The following method invocation would be flagged twice as a violation, once for the extra space before the first comma, and again for the missing space after the second comma:

```
merge(result , left,right);
```

**Space Around Operators**

**Severity:** Low

**Summary**

Binary operators should be surrounded by white space.

**Description**

This audit rule checks for uses of binary operators in which there is no white space on either the left or right side of the operator.

If checked, the "Ignore prefix operators" option causes the audit rule to ignore prefix operators when checking for whitespace violations (e.g. do not flag the expression "(-i)" as a violation).

The "Ignore postfix operators" option is similar (e.g. do not flag the expressions "(i++)" or i++; as violations).

**Example**

The following arithmetic expression would be flagged twice as a violation, once for the missing space before the operator and again for the missing space after the operator:

```
index+1
```

**Space Around Periods**

**Severity:** Low

**Summary**

Periods should not be surrounded by white space.

**Description**

This audit rule checks for uses of periods (within qualified names) in which there is white space on either the left or right side of the period.

**Example**

The following method invocation would be flagged twice as a violation, once for the space before the period and again for the space after the period:

```
point . getX()
```

**Specify an Error Page**

**Severity:** Medium

**Summary**

JSP pages should specify an error page.

**Description**

This audit rule finds JSP files that do not contain a page directive that specifies the error page to use.

**Specify Schema Version**

**Severity:** Medium

**Summary**

Version number must be part of the schema definition.

**Description**

This audit rule looks for schema declarations that do not have the `version` attribute specified. The version number in the schema attribute is not validated against, but will provide a consistent way to determine the version of a particular schema. This version number does not carry over to the instance documents.

**Example**

The following schema declaration does not specify version attribute and would thus be marked as a violation:

```
<schema
targetNamespace="http://example.com/stockquote/v1"
xmlns="http://www.w3.org/2000/10/XMLSchema">
...
</schema>
```

**Specify SOAP Actions For WSDL Operations**

**Severity:** Medium

**Summary**

A WSDL must contain a SOAP Action for all defined operations.

**Description**

This audit rule looks for `wsdlsoap:operation`s that do not specify a `soapAction` attribute.

**Example**

The following operation declaration does not specify `soapAction` and would thus be marked as a violation:

```
<wsdlsoap:operation>
...
</wsdlsoap:operation>
```

**Spell Check Comments**

**Severity:** Low

**Summary**

Comments should be spelled correctly.

**Description**

This audit rule finds comments that contain misspelled words.

**Spell Check Identifiers**

**Severity:** Low

**Summary**

Identifiers should be composed of valid words.

**Description**

This audit rule finds identifiers that are composed from pieces that do not form words.

**Spell Check Property Comments**

**Severity:** Low

**Summary**

Property comments should be spelled correctly.

**Description**

This audit rule finds property comments that contain misspelled words.

**Spell Check Property Names**

**Severity:** Low

**Summary**

Property names should be composed of valid words.

**Description**

This audit rule finds property names that are composed from pieces that do not form words.

**Spell Check Property Values**

**Severity:** Low

**Summary**

Property values should be correctly spelled.

**Description**

This audit rule finds property values that contain misspelled words.

**Spell Check String Literals**

**Severity:** Low

**Summary**

String literals should be correctly spelled.

**Description**

This audit rule finds string literals that contain misspelled words.

**Spell Check XML Attribute Names**

**Severity:** Low

**Summary**

XML attribute names should be spelled correctly.

**Description**

This audit rule finds XML attribute names that contain misspelled words.

**Spell Check XML Attribute Values**

**Severity:** Low

**Summary**

XML attribute values should be spelled correctly.

**Description**

This audit rule finds XML attribute values that contain misspelled words.

**Spell Check XML Body Text**

**Severity:** Low

**Summary**

XML body text should be spelled correctly.

**Description**

This audit rule finds XML body text that contain misspelled words.

**Spell Check XML Comments**

**Severity:** Low

**Summary**

XML comments should be spelled correctly.

**Description**

This audit rule finds XML comments that contain misspelled words.

**Spell Check XML Tag Names**

**Severity:** Low

**Summary**

XML tag names should be spelled correctly.

**Description**

This audit rule finds XML tag names that contain misspelled words.

**SQL Injection**

**Severity:** High

**Summary**

SQL queries might be receiving data from the user or other unsafe sources.

**Description**

SQL Injection occurs when the user is able to enter data directly into SQL queries.

To detect violations, this audit rule searches the code for SQL queries such as `java.sql.Statement.execute(..)` and traces where the query data could have come from. In cases where the source of the query is user input, such as data from a servlet request, `javax.servlet.ServletRequest.getParameter(java.lang.String)`, or from a SWT Text widget, `org.eclipse.swt.widgets.Text.getText()`, a violation is created.

These two sets of methods, the locations where tainted user data can come from and the methods used to query the database, are editable by the user.If methods are missing that are in a common package (such as `java.lang.*`), please let CodePro support know.

Also note, the SQL query methods for Hibernate and Persistence frameworks have been added.

**Security Implications**

Successful SQL Injection attacks can potentially drop tables, update the database in a malicious manner and even gain administrator access.

**Example**

The invocation of the method `executeQuery(..)` would be flagged as a violation since it uses the first name information passed from a servlet request:

```
ServletRequest servletRequest;
Connection connection;
Statement statement;

servletRequest = ...;
connection = DriverManager.getConnection("www.example.com", "myUserName",
"myPassword");
statement = connection.createStatement();
```

```
String firstName = req.getParameter("firstName");
String query = "SELECT * FROM user_data WHERE firstName = '" + firstName +
"'";
statement.executeQuery(query);
```

**Start Method Invoked In Constructor**

**Severity:** Medium

**Summary**

Do not invoke `java.lang.Thread.start()` method inside constructors.

**Description**

This audit rule looks for invocations of `java.lang.Thread.start()` method inside constructors.

**Security Implications**

If the class is extended/subclassed, the method `start()` will be invoked before the constructor of the subclass has finished, potentially allowing the new thread to see the object in an inconsistent state.

**Example**

The following invocation of the `start` method will be marked as a violation because it is called inside the constructor:

```
public class SimleClass {
public SimleClass() {
new Thread() {
public void run(){
}
}.start();
}
}
```

**Statement Creation**

**Severity:** Medium

**Summary**

SQL statements should be prepared or callable.

**Description**

This audit rule checks for the use of any of the createStatement methods defined in java.sql.Connection. These methods should not be used to create statements because the statements might not be either prepared or callable.

**Example**

The following method invocation would be flagged as a violation:

```
connection.createStatement();
```

**Static Field Naming Convention**

**Severity:** Medium

**Summary**

Static fields should have names that conform to the defined standard.

**Description**

This audit rule checks the names of all static fields that are not also final.

**Example**

If the rule has been configured to allow only names that begin with an upper case letter, then the following declaration would be flagged as a violation:

```
public static int minutesPerHour = 60;
```

**Static Field Security**

**Severity:** Medium

**Summary**

Refrain from using non-final public static fields.

**Description**

To the extent possible, refrain from using non-final public static fields because there is no way to check whether the code that changes such fields has appropriate permissions.

**Security Implications**

Public, non-final static fields are accessible and changeable from anywhere within the application making them potential targets of malicious users.

**Example**

The following declaration would be flagged as a violation because it is both public and non-final:

```
public static int minutesPerHour = 60;
```

**Static Instantiation**

**Severity:** Medium

**Summary**

Do not instantiate classes which contain only static methods.

**Description**

Instead of instantiating a class in order to call a static method, you should simply call Class.method(). This saves memory by not creating useless instances of classes.

**Example**

Given two classes, `Foo` and `Bar`, the following would be flagged as a violation:

```
public class Foo {
public static doSomething() {
...
}
}

public class Bar {
public doSomethingElse() {
new Foo().doSomething();
```

```
}
}
```

**Static Member Access**

**Severity:** Medium

**Summary**

Static members should only be accessed by referencing the type in which they are declared.

**Description**

This audit rule looks for references to static members that use either a subtype of the type in which they are declared or an instance of either the declaring type or one of its subtypes.

**Example**

Given the following declarations:

```
public class DeclaringClass
{
public static final int ZERO = 0;
...
}

public class SubclassOfDeclaringClass extends DeclaringClass
{
...
}

DeclaringClass instanceOfDeclaringClass;
SubclassOfDeclaringClass instanceOfSubclass;
```

The following expressions would all be flagged as violations:

```
SubclassOfDeclaringClass.ZERO
instanceOfDeclaringClass.ZERO
instanceOfSubclass.ZERO
```

**String Comparison**

**Severity:** Medium

**Summary**

Strings should not be compared using equals (==) or not equals (!=).

**Description**

Strings should always be compared using one of the comparison methods defined for strings. This audit rule looks for comparisons using either the equals (==) or not equals (!=) operators.

**Example**

```
String currentName, proposedName;

...
if (proposedName != currentName) {
...
}
```

**String Concatenation**

**Severity:** Medium

**Summary**

Strings should not be concatenated.

**Description**

This audit rule finds places where two Strings are concatenated. Concatenation of localized strings to produce a longer string is not valid because the word order can change from one locale to the next. Instead, the class MessageFormat should be used to construct the string based on a pattern that can be localized for each locale.

**Example**

The following expression would be flagged as a violation:

```
return "The button was clicked " + clickCount + " times.";
```

**String Concatenation in Loop**

**Severity:** Medium

**Summary**

Strings should not be concatenated within a loop.

**Description**

The code to concatenate two strings is not very efficient because it creates a StringBuffer for each concatenation. When placed in a loop, this can result in the creation and collection of large numbers of temporary objects. You can create the StringBuffer before entering the loop, and append to it within the loop, thus reducing the overhead.

**Example**

```
String[] path;
String result = "";
for (int i = 0; i < path.length; i++) {
result = result + "." + path[i];
}
```

**String Created from Literal**

**Severity:** Medium

**Summary**

Strings should not be created from a String literal.

**Description**

This audit rule finds places in the code where a String literal is used to initialize a newly created String. Doing so is almost never necessary and usually only serves to waste both time and space.

**Example**

The following expression would be flagged as a violation:

```
new String("Pause");
```

**String indexOf Use**

**Severity:** Medium

**Summary**

Don't compare output from String.indexOf with > 0 or <= 0.

**Description**

This audit rule looks for the common off-by-one-error caused by comparing String.indexOf() to 0, for example: "indexOf(..) > 0." This is read as "if indexOf is greater than 0 then the there doesn't exist an instance of what we are looking for", but the mistake here is that indexOf returns -1 if nothing was found, not 0. Hence, the user meant ">=", not ">". The opposite mistake is made with "indexOf <= 0."

All "indexOf" methods in java.lang.String are detected by this rule, see indexOf(int), indexOf(int, int), lastIndexOf(int), lastIndexOf(int, int), indexOf(String), etc.

**Example**

The following comparison would be flagged as a violation:

```
str.indexOf('.') > 0
```

**String Literals**

**Severity:** Medium

**Summary**

String literals should not appear in code.

**Description**

This audit rule checks for string literals within the code that are not included in a user-defined list of acceptable literal values. The rule can be configured to ignore strings that are used as static final field initializers, contain single characters, only whitespace and/or digits and/or a specially formed trailing comment.

The primary purpose of this rule is to aid in internationalization efforts. With this rule on, you can identify all of the hardcoded strings in the application and move them out into external property files that can be translated.

**Example**

```
String message = "Houston, we have ignition.";
```

**String Method Usage**

**Severity:** Low

**Summary**

The String.equals(), String.equalsIgnoreCase() and String.comparesTo() methods should not be used in an internationalized environment.

**Description**

This audit rule checks for the use of the String.equals(),String.equalsIgnoreCase(), String.comparesTo() and String.indexOf() methods in an internationalized environment. These methods cannot be relied on to sort strings because the Unicode values of the characters in the strings do not correspond to the relative order of the characters in most languages.

**Example**

The following expression would be flagged as a violation:

```
if (stringArray[i].equals(stringToInsert)) {
...
}
```

**StringTokenizer Usage**

**Severity:** Low

**Summary**

The StringTokenizer class should not be used in an internationalized environment.

**Description**

This audit rule checks for the use of the StringTokenizer class in an internationalized environment. This class cannot be relied on because the Unicode values of the characters in the strings do not correspond to the relative order of the characters in most languages.

**Example**

The following instance creation expression would be flagged as a violation:

```
new StringTokenizer(inputString)
```

**Struts Validator Not in Use**

**Severity:** Low

**Summary**

Use the Struts validator to prevent vulnerabilities that result from unchecked input.

**Description**

Unchecked input is the leading cause of vulnerabilities in J2EE applications. To prevent such attacks, use the Struts validator to check all program input before it is processed by the application. To use the Struts validator you should configure it in struts-config.xml:

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
...
</plug-in>
```

**Example**

The tag <struts-config> would be flagged if it does not contain a ValidatorPlugIn declaration:

```
<struts-config>
...
<struts-config>
```

**Subclass should override method**

**Severity:** Medium

**Summary**

If you want to override a method declared in a superclass, you should not change the signature of a method which belongs to the subclass.

**Description**

This rule looks for places where a method of defined in a subclass has the same name as a method defined in a superclass but a different signature.

**Security Implications**

If you want to override the method in the superclass, you should change signature method which belongs to subclass.

**Example**

The following invocation method which belongs to subclass will be flagged as a violation because a signatures do not match.

```
public class Parent
{
public void func(int a) {
.........
}
} public class Child extends Parent
{
public void func(double a) {
...
}
}
```

**Synchronization On getClass Method**

**Severity:** Medium

**Summary**

You should avoid synchronization on `getClass()` method.

**Description**

This rule looks for places where the result of the `getClass()` method is used for synchronization.

**Security Implications**

If this class is subclassed, subclasses will synchronize on a different class object, which isn't likely what was intended.

**Example**

The following code is synchronized on `getClass` method and thuswill be marked as a violation:

```
public void methodOne(){
synchronized ( getClass() ){
}
}
```

**Synchronization On Non Final Fields**

**Severity:** Medium

**Summary**

All fields used for synchronization of threads should be declared final.

**Description**

This rule looks for places where blocks are synchronized on non-final fields.

**Security Implications**

If an attacker gets access to the field which is used for synchronization, he could modify it and create an error in synchronization logic. Such vulnerability could result in an application misbehaviour or crash or access to otherwise protected data.

**Example**

The following code will be flagged as a violation because the field that is used for synchronization is not final.

```
public class Test {
Object lock;
SOUF(Object lock) {
this.lock = lock;
}
public void myMethod() {
synchronized(lock) {
...
}
}
}
```

**Synchronized In Loop**

**Severity:** Medium

**Summary**

Synchronized statement used in a loop.

**Description**

This audit rule looks for uses of the synchronized statement that occur within a loop. Synchronization is relatively expensive, so the synchronized statement should be moved to enclose the loop.

**Example**

The following use of the synchronized statement would be flagged as an error:

```
for (int i = 0; i < 10; i++) {
synchronized (monitor) {
monitor.doSomeWork();
}
}
```

**Synchronized Method**

**Severity:** Medium

**Summary**

Methods should never be marked as synchronized.

**Description**

This audit rule reports the use of the synchronized modifier with methods. It is too easy to miss the existence of the synchronized modifier, so following this rule leads to more readable code.

**Example**

```
public synchronized String getName()
{
```

```
...
}
```

**Tag Handler Field Not Cleared**

**Severity:** Medium

**Summary**

Checks for custom tag handler fields that have not been cleared in release().

**Description**

A practice recommended by the JSF tutorial is to override the release() method and clear non-final instance fields in every custom tag handler. This audit rule looks for custom tag handlers that have non-final instance fields of non-primitive types that are not cleared in the release() method.

**See Also**

The audit rule "Tag Handler Should Implement Release" checks for the existance of the release() method in classes that need it.

**Example**

The field "bar" of the following class will be marked is violation because it is not assigned to within the release() method.

```
public class MyTag extends UIComponentClassicTagBase {
private String foo;
private String bar;
public void release() {
foo = null;
}
}
```

**Tag Handler Should Implement Release**

**Severity:** Medium

**Summary**

Custom tag handlers should implement the release() method.

**Description**

A practice recommended by the JSF tutorial is to override the release() method and clear non-final instance fields in every custom tag handler. This audit rule looks for custom tag handlers that have non-final instance fields of non-primitive types that are not cleared in the release() method.

**See Also**

The audit rule "Tag Handler Field Not Cleared" checks for fields that should be cleared in the release() method but are not.

**Example**

The following class will be flagged as a violation because it has a field to release but does not override the release() method.

```
public class MyTag extends UIComponentClassicTagBase {
private String foo;
}
```

**Tainted Filter**

**Severity:** High

**Summary**

Request parameters and other tainted data should not be passed into LDAP context `search()` filter without sanitizing.

**Description**

This audit rule violates the usage of unvalidated user input in LDAP search request filters.

**Security Implications**

Only trusted data should be used in LDAP search filter, otherwise an attacker can create a malformed request similar to the classic SQL injection. This can lead to revealing otherwise protected data.

**Example**

The following code receives data via `HttpServletRequest#getParameter()` call and does not clean it before putting it into the LDAP search filter:

```
LdapContext ctx = getDirContext();
String userName = req.getParameter("user");
String filter = "(user=" + userName + ")";
try {
NamingEnumeration users = ctx.search("ou=People,dc=example,dc=com", filter,
null);
```

**Tainted Internet Address**

**Severity:** High

**Summary**

Request parameters and other tainted data should not be passed into methods creating an InternetAddress without sanitizing.

**Description**

This rule looks for places where tainted user input is used as a part of the string used in creating an InternetAddress.

**Security Implications**

Only trusted data should be used in an InternetAddress, otherwise an attacker can perform a URL injection attack, potentially redirecting the system to access untrusted data.

**Example**

The following code uses user input directly as an InternetAddress:

```
String address = req.getParameter("address");
InternetAddress addr = new InternetAddress(url);
```

**Temporary Object Creation**

**Severity:** Low

**Summary**

Instances of numeric classes should not be created solely for the purpose of converting a numeric value to a string.

**Description**

This audit rule checks for the creation of numeric classes where the only purpose for the object is to invoke the toString() method on it. All of the numeric classes implement a static toString() method that can do the same thing, but without the cost of creating and collecting an extra object.

**Example**

The following expression would be flagged as a violation:

```
(new Integer(age)).toString()
```

**Test Case Naming Convention**

**Severity:** Medium

**Summary**

Test case names should conform to the defined standard.

**Description**

This audit rule checks the names of all test cases to ensure that they conform to the standard.

**Example**

If the rule were configured to require that the word "Test" be appended to every test case's name, the following class declaration would be flagged as a violation:

```
public class MyTestClass extends TestCase
{
...
}
```

**Throw in Finally**

**Severity:** Medium

**Summary**

Finally blocks should not contain a throw statement.

**Description**

This audit rule finds places where a throw statement is contained in a finally block.

**Example**

The following throw statement would be flagged as a violation because it occurs within a finally block:

```
try {
...
} finally {
throw new Exception("This is never OK");
}
```

**Thrown Exceptions**

**Severity:** Medium

**Summary**

Some exceptions should not be thrown.

**Description**

This audit rule finds throw statements that throw an exception class that is disallowed. The list initially includes exception classes that are either too general (such as Throwable or Exception), or that are unchecked (Error, RuntimeException, and all subclasses of either).

**Example**

If the rule is configured to disallow throwing of instances of Throwable, then the following throw statement would be flagged as a violation:

```
throw new Throwable("Bet this won't get caught");
```

**toString() Method Usage**

**Severity:** Low

**Summary**

The toString() methods for Dates, Times and Numerics should not be used in an internationalized environment.

**Description**

This audit rule checks for the use of the toString() methods for Dates, Times and Numerics in an internationalized environment. Date, time and numeric formats differ with region and language, so consistent results cannot be counted on.

**Example**

The following use of the toString() method would be flagged as a violation:

```
today = new Date();
dateString = today.toString();
```

**Transient Field in Non-Serializable Class**

**Severity:** Medium

**Summary**

Classes that do not implement java.io.Serializable should not declare fields as transient.

**Description**

This audit rule looks for declarations of fields that are transient in classes that do not implement the interface java.io.Serializable.

**Example**

The following field would be flagged as a violation:

```
class NonSerializable
{
private transient int value;
}
```

**Type Declarations**

**Severity:** Low

**Summary**

Verification that type declarations follow a specified style of coding.

**Description**

This audit rule finds source in which type declarations do not follow a specified style of coding. One example is that the first type declaration in a file should be a one with a name matching that of the file if such a type is defined in the file.

**Type Depth**

**Severity:** Medium

**Summary**

Types should not be deeply nested.

**Description**

This audit rule finds source in which type declarations are deeply nested. The maximum depth can be configured. Top level types have a depth of zero (0), so a maximum depth of 2 would mean, for example, that an inner class of a top-level class could have an anonymous inner class without being flagged, but a third level of nesting would be flagged.

**Type Javadoc Conventions**

**Severity:** Low

**Summary**

All types should have a Javadoc comment associated with them.

**Description**

This audit rule checks for the existence of a Javadoc comment for each type. It optionally checks

for the existence of at least one @author tag and always ensures that every author tag has some text following it. It also optionally checks for the existence of at least one @version tag and always ensures that every version tag has some text following it.

**Type Member Ordering**

**Severity:** Low

**Summary**

Verification that type members follow a specified order.

**Description**

This audit rule is used to check the order of members within a type. Simple rules include checking that main() is last and that members with a common name are grouped together. More complex orderings are specified with the Global Ordering check box. When the global ordering check box is checked you can choose one of the orderings from the list:

Alphabetical - all members should be arranged alphabetically by name with no regard for member type or modifiers. Initializers have no names and can appear anywhere.

Fields First - members should appear in the order: fields, initializers, constructors, methods, inner types. Within those groups they should be arranged alphabetically.

Constructors First - members should appear in the order: constructors, fields, initializers, methods, inner types. Within those groups they should be arranged alphabetically.

Public to Private - members should appear in the modifier order: public, protected, private. Within those groups they should be arranged alphabetically.

Private to Public - members should appear in the order: private, protected, public. Within those groups they should be arranged alphabetically.

The above choices are followed by combinations. "Fields First/Public to Private", for example, means that members should be arranged in Fields First order, be arranged in Public to Private order within those groups, then alphabetically. The other combinations follow suit.

**Type Names Must Be Singular**

**Severity:** Medium

**Summary**

Type names must be singular.

**Description**

This audit rule checks for types whose name is plural.

**Example**

The following class declaration would be flagged because the name is plural:

```
public class Employees
{
...
}
```

**Type Parameter Naming Convention**

**Severity:** Medium

**Summary**

Type parameter names should conform to the defined standard.

**Description**

This audit rule checks the names of all type parameters.

**Example**

If the rule were configured to require that all type parameter names start with a capital letter, the following declaration would be flagged as a violation:

```
public class BetterCollection<elementType> ...
```

**Unassigned Field**

**Severity:** Medium

**Summary**

Private fields should be assigned a value.

**Description**

This audit rule looks for private fields that are never assigned a value.

**Undefined Message Key**

**Severity:** Medium

**Summary**

Message keys must be defined in the resource file.

**Description**

This audit rule looks for uses of message keys that are not defined in the resource file. All messages created in the Action and ActionForm classes should be declared in the special resource file, which can be found with the Struts configuration file.

**Example**

In the following code, the name of the message key would be flagged if the key "messages.invaliduser" did not exist in the resource file.

```
public ActionForward execute(ActionMapping mapping, ActionForm inForm,
HttpServletRequest request, HttpServletResponse response) throws Exception {
...
{
ActionMessages messages = new ActionMessages();
messages.add(ActionMessages.GLOBAL_MESSAGE,
new ActionMessage("messages.invaliduser"));
saveErrors(request, messages);
return mapping.findForward("fail");
}
...
}
```

**Undefined Placeholder**

**Severity:** Medium

**Summary**

All properties used in placeholders should be defined in the property file.

**Description**

This audit rule looks for properties used in placeholders that are not defined in the associated property file.

**Example**

The string jdbc.missed would be flagged if "jdbc.missed" doesn't exist in the property file.

```
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer
">
<property name="location">
<value>/WEB-INF/jdbc.properties</value>
</property>
</bean>

<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName">
<value>${jdbc.driver}</value>
</property>
<property name="url">
<value>${jdbc.url}</value>
</property>
<property name="username">
<value>${jdbc.user}</value>
</property>
<property name="missed">
<value>${jdbc.missed}</value>
</property>
</bean>
```

**Undefined Property**

**Severity:** Medium

**Summary**

Properties referenced in a plugin.xml file need to be defined in the corresponding plugin.properties file.

**Description**

The plugin.xml file can be internationalized by using properties names (names prefixed by a percent sign (%)) anywhere a value is expected This rule checks to see that all referenced properties are defined in the corresponding plugin.properties file.

**Unhashable class in hashed collection**

**Severity:** High

**Summary**

In order to be stored in a hashed collection, a class must override both the `equals()` method, and the `hashCode()`, or else it must not override either.

**Description**

This audit rule finds places where a type that only overrides one of `equals()` or `hashCode()` is used in a hashed collection.

**Unique Namespace per Schema**

**Severity:** Medium

**Summary**

Every schema should have a unique namespace.

**Description**

Target namespaces must be heterogeneous. This means that every schema will have a unique namespace. While this will not allow the use of 'redefine', it will ensure that no data collisions occur. Make sure to use groupings to ensure your schema namespace is unique.

**Example**

Two schemas with the same targetNamespace will be marked as a violation:

```
...
<schema
targetNamespace="http://faulty-domain.com/stockquote.xsd"
xmlns="http://www.w3.org/2000/10/XMLSchema">
...
```

```
</schema>
...
<schema
targetNamespace="http://faulty-domain.com/stockquote.xsd"
xmlns="http://www.w3.org/2000/10/XMLSchema">
...
</schema>
```

**Unknown Cast**

**Severity:** Medium

**Summary**

When an object is checked for being an instance of one class and then cast to another class that is neither inherited from the checked class nor its ancestor this usually results in a ClassCastException.

**Description**

This audit rule violates class casts that are preceded with a check of the same instance for being intanceof of the incomparable class.

**Security Implications**

Such a cast is usually an error which results in a `ClassCastException`. This may result in an unexpected behavior of an application that could possibly compromise its security.

**Example**

The following code would be flagged as a violation because it performs a check for one class and then casts the instance on an incomparable class:

```
if (s instanceof String) {
StringBuffer db = (StringBuffer) s;
...
}
```

**Unknown Component Type**

**Severity:** Medium

**Summary**

The method getComponentType should return the name of a declared component type.

**Description**

This audit rule looks for subclasses of UIComponentTagBase that implement a getComponentType method that returns a constant value that is not mentioned in faces-config.xml. Because the JSF configuration file lists all component types in the project, there is no point in returning a component type that is not defined in faces-config.xml.

**Example**

A violation will be signalled if "RoundButton" component type is not defined in the configuration file.

```
public class RoundButtonTag extends UIComponentTag {
...
public String getComponentType() {
return "RoundButton";
}
...
}
```

**Unknown Renderer Type**

**Severity:** Medium

**Summary**

The method getRendererType should return the name of a declared renderer.

**Description**

This audit rule looks for subclasses of UIComponentTagBase that implement a getRendererType method that returns a constant value that is not mentioned in faces-config.xml. Because the JSF configuration file lists all renderers in the project, there is no point in returning a renderer type that is not defined in faces-config.xml.

**Example**

A violation will be flagged if the "RoundButton" renderer is not defined in the configuration file.

```
public class RoundButtonTag extends UIComponentTag {
...
public String getRendererType() {
return "RoundButton";
}
...
}
```

**Unnecessary "instanceof" Test**

**Severity:** Medium

**Summary**

Unnecessary instanceof tests should be removed.

**Description**

This audit rule looks for unnecessary uses of "instanceof". An "instanceof" test against a superclass or superinterface of the static type of an object is unnecessary and should be removed.

**Example**

```
"this is a string" instanceof String;
```

**Unnecessary Catch Block**

**Severity:** High

**Summary**

Catch blocks should do more than rethrow the exception.

**Description**

This audit rule looks for catch blocks whose body consists only of a throw of the exception that was just caught. Unless there is another catch block in the same try statement that catches a superclass of the exception's class, such catch blocks only serve to clutter the code.

**Example**

The following catch clause would be flagged as a violation:

```
try {
...
} catch (NullPointerException exception) {
throw exception;
}
```

**Unnecessary Default Constructor**

**Severity:** High

**Summary**

Default constructors should not be declared unless necessary.

**Description**

This audit rule looks for declarations of default (zero-argument) constructors in classes with no other constructors, whose body is either empty or contains only an invocation of the superclass' default constructor.

**Example**

The following constructor would be flagged as a violation:

```
public UnnecessaryConstructor() {
super();
}
```

**Unnecessary Exceptions**

**Severity:** Medium

**Summary**

Unnecessary exceptions should be removed.

**Description**

This audit rule checks for methods that declare an exception that cannot be thrown within the body of the method.

There are two options. The first controls whether unchecked exceptions will be allowed to be declared. This is sometimes desirable to allow the full range of exceptions to be fully documented.

The second option controls whether a class of exception can be declared when a subclass of the exception class is thrown, or whether only the classes of exceptions that are actually thrown can be declared.

Note: this rule does not examine the implementations of a method that occur in subclasses to see whether an exception is being declared in a superclass in order to allow it to be thrown by a method in a subclass.

**Example**

The following throws clause would be flagged as a violation because there is no way for the exception to be thrown:

```
public int getChildCount()
throws RemoteException
{
return 0;
}
```

**Unnecessary Final Method**

**Severity:** Medium

**Summary**

Methods in a final class should not be declared as final.

**Description**

This audit rule looks for methods that are declared as being final that are defined in a class that is also declared as final. It is not necessary to declare the method to also be final because no subclass of the class could ever be defined.

**Example**

The "final" modifier on the following method would be flagged as a violation because the class is also marked as being final"

```
public final class Point
{
```

```
public final int getX()
{
...
}
}
```

**Unnecessary Import Declarations**

**Severity:** Medium

**Summary**

There should not be imports for types or packages that are not referenced.

**Description**

This audit rule checks to ensure that each of the import declarations is actually necessary. A type import is considered necessary if the type that is imported by it is referenced within the compilation unit. A demand import is considered necessary if there is at least one type within the specified package that is referenced that is not also imported by a type import.

In neither case does this rule consider whether the reference to the type is fully qualified, only that the type is being referenced.

**Example**

The following import statement would be flagged as a violation because the package is always implicitly imported:

```
import java.lang.*;
```

**Unnecessary Null Check**

**Severity:** Medium

**Summary**

A variable is being checked against null when it is not necessary.

**Description**

This rule identifies places where an object-valued variable is being compared to null when the comparison is unnecessary because of preceding code.

**Example**

The following comparison would be flagged as a violation:

```
airplane.prepareForTakeOff();
if (airplane != null) ...
```

**Unnecessary Override**

**Severity:** Medium

**Summary**

Methods that override other methods should do more than invoke the overridden method.

**Description**

This audit rule looks for methods whose body consists only of an invocation of the overridden method with the same argument values. Such methods can safely be removed.

**Example**

The following method would be flagged as a violation because it only invokes the method it overrides:

```
public void clear() {
super.clear();
}
```

**Unnecessary Return**

**Severity:** Medium

**Summary**

Methods that do not return a value should not end with a return.

**Description**

This audit rule finds methods that are declared to not return a value (void) but whose last statement is a return statement.

**Example**

```
public void markChanged()
{
changed = true;
return;
}
```

**Unnecessary Return Statement Parentheses**

**Severity:** Low

**Summary**

The expression in a return statement should not be parenthesized.

**Description**

This audit rule looks for return statements where the expression that computes the value to be returned is enclosed in parentheses. They are unnecessary and typically make the code harder to read and maintain.

**Example**

The following return statement's expression would be flagged as a violation because the parentheses are not necessary:

```
return (x * x);
```

**Unnecessary toString() Method Invocation**

**Severity:** Medium

**Summary**

Remove unnecessary invocations of toString().

**Description**

This audit rule flags instances of toString() that are called on String objects. Removing such invocations does not affect the program logic and can reduce timing.

**Example**

The invocation of toString() below would be flagged as it can be removed.

```
String string = ...;
System.out.println(string.toString());
```

**Unnecessary Type Cast**

**Severity:** Medium

**Summary**

Unnecessary type casts should be removed.

**Description**

This audit rule checks for places where a value is being cast to another type and the type cast is not necessary. This includes the following cases:
- casting from one type to the same type,
- casting from one type to a supertype of that type,
- casting to a more specific type when the result will be assigned to a variable of the same type, or
- casting immediately prior to using the instanceof operator to test the type.

**Example**

The following cast would be flagged because the type of the literal is already int:

```
int i = (int) 0;
```

The following cast would be flagged because the variable list can be assigned to the variable collection without the cast:

```
List list = new ArrayList();
Collection collection = (ArrayList) list;
```

**Unregistered Validator**

**Severity:** Medium

**Summary**

All implementations of the Validator interface should be registered in the configuration file.

**Description**

This audit rule looks for non-abstract implementations of the Validator interface that are not registered in faces-config.xml. Implementations that are not registered will never be used by the framework.

**Example**

The following class will be marked as violation if it is not registered in the configuration file.

```
public class UnregisteredValidator implements Validator {
public void validate(FacesContext context, UIComponent comp, Object value) {
...
}
}
```

**Unsupported Clone**

**Severity:** Medium

**Summary**

Find invocations of clone() that will throw an exception.

**Description**

This audit rule checks for invocations of the clone() method on objects that do not either implement the Cloneable interface or override the default implementation of clone() in Object.

**Example**

The following invocation would be flagged as being a violation if the object is not Cloneable:

```
object.clone();
```

**Unused Assignment**

**Severity:** Medium

**Summary**

Local variable or parameter is assigned a value that is never used.

**Description**

This rule violates cases when local variable or parameter is assigned a value that is either reassigned or simply not used till the end of the scope.

**Security Implications**

Unused result of an assignment usually indicates inconsistency in the code, possible errors or poorly mantained code, which may result in a security threat.

**Example**

The following method reassignes a parameter without using its initial value. This is probably a misprint or an outdated code and should be removed:

```
public void testReassigned(int i) {
i = 10;
...
}
```

**Unused Field**

**Severity:** Medium

**Summary**

Fields that are not used should be removed.

**Description**

This audit rule checks for any private instance fields that are not referenced within their declaring class.

**Example**

The following field would be flagged as a violation if it is not used within its declaring class:

```
private int unused;
```

**Unused Label**

**Severity:** Medium

**Summary**

Labels that are not used should be removed.

**Description**

This audit rule checks for any labeled statements whose labels are not used in either a break or continue statement within the scope of the labeled statement.

**Example**

The following label would be flagged as a violation because it is not used within the body of the for loop:

```
int sum = 0;
sumElements: for (int i = 0; i < array.length; i++) {
sum = sum + array[i];
}
```

**Unused Method**

**Severity:** Medium

**Summary**

Methods that are not used should be removed.

**Description**

This audit rule checks for any private instance methods that are not referenced within their declaring class.

**Example**

The following method would be flagged as a violation if it is not used within its declaring class:

```
private int unused()
{
...
}
```

**Unused Return Value**

**Severity:** Medium

**Summary**

The value returned from methods should be used.

**Description**

This audit rule looks for invocations of methods that return values where the value is ignored. Most methods that return a value either have no side-effect or are using the returned value as an indication of success or failure. In the first case, the invocation should be removed if the value is not needed. In the second case, the status value should be checked.

**Example**

The following method invocation would be flagged as a violation if the method `getX` returns a value:

```
point.getX();
```

**Unused StringBuffer**

**Severity:** Medium

**Summary**

The contents of a StringBuffer should be used.

**Description**

This audit rule checks for any instances of the class StringBuffer whose contents are not retrieved. This usually means that the code to use the contents of the buffer was omitted, but can also indicate that old code is no longer needed and should have been deleted.

**Example**

The StringBuffer declared in the following method would be flagged as a violation:

```
private String toString()
{
StringBuffer buffer;

buffer = new StringBuffer();
buffer.append("Product #");
buffer.append(getName());
return getName();
}
```

**Unused StringBuilder**

**Severity:** Medium

**Summary**

The contents of a StringBuilder should be used.

**Description**

This audit rule checks for any instances of the class StringBuilder whose contents are not retrieved. This usually means that the code to use the contents of the builder was omitted, but can also indicate that old code is no longer needed and should have been deleted.

**Example**

The StringBuilder declared in the following method would be flagged as a violation:

```
private String toString()
{
StringBuilder builder;

builder = new StringBuilder();
builder.append("Product #");
builder.append(getName());
return getName();
}
```

**Unused Validation Form**

**Severity:** Low

**Summary**

An unused validation form indicates that validation logic is not up-to-date.

**Description**

It is easy for developers to forget to update validation logic when they remove or rename action form mappings. One indication that validation logic is not being properly maintained is the presence of an unused validation form.

**Example**

If the struts-config.xml file contains:

```
<form-beans>
<form-bean name="login" type="example.LoginForm"/>
<form-bean name="validateForm" type="example.ValidatedForm"/>
</form-beans>
```

and the validation.xml file contains

```
<form-validation>
<formset>
<form name="login"/>
<form name="validatedForm"/>
</formset>
</form-validation>
```

the entry for "validatedForm" would be flagged in the validation.xml file because it is not defined in struts-config.xml (note the subtle name difference).

**Unvalidated Action Field**

**Severity:** Medium

**Summary**

All fields of a Struts2 action should be validated even if some of them are unused.

**Description**

In situations when one Struts2 action class is used in several forms and each form uses a subset of the classes fields it is a common mistake to validate only those fields which are used in a given form.

**Security Implications**

This may allow an attacker to inject unvalidated input into your application. Thus, all fields of a form should be validated even if some of them are unused.

**Example**

Let there be a Struts2 action class, `QuizAction`, that declares three fields, `name`, `age` and `answer`. The following entry in the `QuizAction-validation.xml` will lead to a violation because it validates only two of them:

```
<validators>
<field name="name">
<field-validator type="requiredstring">
<message>You must enter a name</message>
</field-validator>
</field>
<field name="age">
<field-validator type="int">
<param name="min">13</param>
<param name="max">19</param>
<message>Only people ages 13 to 19 may take this quiz</message>
</field-validator>
</field>
</validators>
```

**Unvalidated Action Form**

**Severity:** Low

**Summary**

Every action form must have a corresponding validation form.

**Description**

If a Struts action form mapping specifies a form, it must have a validation form defined under the Struts validator. If an action form mapping does not have a validation form defined, it may be vulnerable to a number of attacks that rely on unchecked input.

**Example**

If the validation.xml file contains

```
<form-validation>
<formset>
<form name="login"/>
<form name="unvalidatedForm"/>
</formset>
</form-validation>
```

and struts-config.xml file contains:

```
<form-beans>
<form-bean name="login" type="example.LoginForm"/>
<form-bean name="validatedForm" type="example.ValidatedForm"/>
</form-beans>
```

then the action form validatedForm would be flagged because it's not referenced in validation.xml.

**Unvalidated Action Form Field**

**Severity:** Medium

**Summary**

All fields of a Struts action form should be validated even if some of them are unused.

**Description**

In situations when one Struts action form class is used in several forms and each form uses a subset of the classes fields it is a common mistake to validate only those fields which are used in a given form.

**Security Implications**

This may allow an attacker to inject unvalidated input into your application. Thus, all fields of a form should be validated even if some of them are unused.

**Example**

Let there be a Struts form class, `QuizAction`, that declares three fields, `name`, `age` and `answer`.

The following entry in the `validation.xml` will lead to a violation because it validates only two of them:

```
<form name="quizForm">
<field property="name" depends="required">
<arg key="quizForm.name"/>
</field>
<field property="age" depends="required">
<arg key="quizForm.age"/>
</field>
</form>
```

**Usage Of Binary Comparison**

**Severity:** Medium

**Summary**

You should use short-circuit operations instead of binary operations.

**Description**

This rule looks for places where a binary operation is used that could be replaced by a short-circuit operator.

**Security Implications**

Usage of binary operation instead of short-circuit can cause unexpected situation when RuntimeException or NullPointerException can be thrown.

**Example**

The following code would be flagged as a violation because a binary and operator (&) is used where a conditional-and operator (&&) could be used:

```
public void func(int[] a)
{
if (a != null & a.length() != 0) {
doSomething();
}
}
```

**Usage Of Static Calendar**

**Severity:** Medium

**Summary**

Do not use `Calendar` objects without synchronization.

**Description**

This rule looks for a places where `Calendar` objects are used without synchronization.

**Security Implications**

`Calendars` are inherently unsafe for multithread usage. If this object is used without proper synchronizations, `ArrayIndexOutOfBoundsException` can be thrown.

**Example**

The following usage of a `Calendar` object will be marked as a violation because it should be used in `synchronized` block:

```
private static Calendar data;
...
public Calendar getCalendar() {
return data;
}
```

**Usage Of Static Date Format**

**Severity:** Medium

**Summary**

Do not use `DateFormat` objects without synchronization.

**Description**

This rule looks for places where a `DateFormat` object is used without synchronization.

**Security Implications**

`DateFormat`s are inherently unsafe for multithread usage. Sharing a single instance across thread boundaries without proper synchronization can result in erratic behavior of the application.

**Example**

The following usage of a `DateFormat` object will be marked as a violation because it should be used in a `synchronized` block:

```
private static DateFormat data;
...
public DateFormat getFormat() {
return data;
}
```

**Use "for" Loops Instead of "while" Loops**

**Severity:** Low

**Summary**

Disallows the use of "while" loops.

**Description**

This audit rule checks for the use of "while" loops rather than "for" loops. "for" loops allow loop-scoped variables to be declared during loop setup, minimizing the scope of loop-control variables to just the body of the loop. Since their scope ends after the loop, they cannot be referenced later accidentally.

**Example**

The following while loop would be flagged as a violation:

```
int index = 0;
while (index < array.length) {
...
index++;
}
```

because it should be rewritten as:

```
for (int i = 0; i < array.length; i++) {
...
}
```

**Use "Nullable" Type for Identifier Properties**

**Severity:** Medium

**Summary**

Use a "nullable" type for identifier properties of persistent classes.

**Description**

Identifier properties of persistent classes should be 'synthetic' (generated, with no business meaning) and of a non-primitive type. This audit rule looks for identifier properties with a primitive type. For maximum flexibility, use java.lang.Long or java.lang.String.

**Example**

The following class would be flagged because it has a primitive-valued identifier field.

```
public class Message {
private int id;
private String text;
private Message nextMessage;

Message() {}

public Message(String text) {
this.text = text;
}

public int getId() {
return id;
}
private void setId(int id) {
this.id = id;
}
}
```

**Use == to Compare With null**

**Severity:** Medium

**Summary**

A null value should not be compared using methods equals() or equalsIgnoreCase().

**Description**

This audit rule finds places where an object is compared to the null value using either the equals() or (if the object is a String) the equalsIgnoreCase() method. In both cases, the contract

of the method requires this comparison to always return false. Either the test is unnecessary, or it should be replaced by an identity comparison.

**Example**

The following comparison would be flagged as a violation:

```
if (object.equals(null))
```

**Use @After Annotation Rather than tearDown()**

**Severity:** Low

**Summary**

Use the @After annotation rather than tearDown() to clean up all the data entities required in running tests.

**Description**

In JUnit 3, the tearDown method was used to clean up all data entities required in running tests. JUnit 4 skips the tearDown method and executes all methods annotated with @After after running each test.

**Example**

The following test case would be flagged as a violation because it uses tearDown() rather than @After for clean up activities:

```
public class MyTest extends TestCase {
public void tearDown(){
doReleaseConnection();
}
}
```

**Use @Before Annotation Rather than setUp()**

**Severity:** Low

**Summary**

Use the @Before annotation rather than setUp() to initialize the data entities required in running tests.

**Description**

In JUnit 3, the setUp method was used to set up all data entities required in running tests. JUnit 4 skips the setUp method and executes all methods annotated with @Before before all tests.

**Example**

The following test case would be flagged as a violation because it uses setUp() rather than @Before for setting up the Junit test:

```
public class MyTest extends TestCase {
public void setUp(){
doEstablishConnection();
}
}
```

**Use @RunWith and @SuiteClasses to build test suite**

**Severity:** Medium

**Summary**

Use the @RunWith and @SuiteClasses annotations rather than suite() method to build test suite for JUnit

**Description**

In JUnit 3, test suites are indicated by the suite() method. In JUnit 4, suites are indicated through the @RunWith and @SuiteClasses annotations.

**Example**

The following test class would be flagged as a violation because it uses suite() rather than @RunWith and @SuiteClasses annotations to construct test suite:

```
public class BadExample extends TestCase{
public static Test suite(){
return new Suite();
```

```
}
}
```

**Use @Test Annotation for JUnit test**

**Severity:** Low

**Summary**

JUnit 4 tests should use the @Test annotation.

**Description**

In JUnit 3, the framework executed all methods which started with the word test as a unit test. In JUnit 4, only methods annotated with the @Test annotation are identified as tests.

**Example**

The following test case would be flagged as a violation because the test method does not contain any @Test annotation:

```
public class MyTest extends TestCase {
public void testFoo(){
doSomething();
}
}
```

**Use a Valid Schema Namespace**

**Severity:** Medium

**Summary**

The schema namespace should start with one of a list of prefixes.

**Description**

This audit rule looks for all schema namespace declarations that do not start with one of a list of preconfigured prefixes. Using such a prefix identifies that the schema belongs to a specific provider.

**Example**

The following service declaration does not use the preconfigured `http://example.com/` URI prefix for `targetNamespace` and would thus be marked as a violation:

```
<schema
targetNamespace="http://faulty-domain.com/stockquote.xsd"
xmlns="http://www.w3.org/2000/10/XMLSchema">
...
</schema>
```

**Use a Valid WSDL Namespace**

**Severity:** Medium

**Summary**

The WSDL namespace should start with one of a list of prefixes.

**Description**

This audit rule looks for all WSDL namespace declarations that do not start with one of a list of preconfigured prefixes. Using such a prefix identifies that the WSDL belongs to a specific provider.

**Example**

The following service declaration does not use the preconfigured `http://example.com/` URI prefix for `tns` namespace and would thus be marked as a violation:

```
<definitions
name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
xmlns:tns="http://error.com/stockquote.wsdl"
xmlns:xsd1="http://example.com/stockquote.xsd"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
...
</definitions>
```

**Use Action Interface**

**Severity:** High

**Summary**

Use the Action interface rather than the Action abstract class.

**Description**

This audit rule looks for actions that subclass the abstract class org.apache.struts.action.Action. They should subclass the interface org.apache.struts2.Action instead.

**Example**

Replace this code

```
public class NewAction extends Action {
public org.apache.struts.action.ActionForward execute(
org.apache.struts.action.ActionMapping mapping,
org.apache.struts.action.ActionForm form,
javax.servlet.http.HttpServletRequest request,
javax.servlet.http.HttpServletResponse response)
throws java.lang.Exception
{
...
return mapping.findForward("forward_name");
}
}
```

with

```
public class NewAction implements Action {
public String execute() {
...
return "forward_name";
}
}
```

**Use Additional Attribute in Tag**

**Severity:** Medium

**Summary**

Use an additional attribute in the <constructor-arg> tag in your configuration file.

**Description**

When you have more than one constructor argument or your class has more than one constructor

you should specify an additional attribute (index or type, prefer type) in the <constructor-arg> tag. This audit rule looks for tags that need an additional argument but do not have one.

**Example**

If you have following class:

```
public class ConstructorConfusion {

private String someValue;

public ConstructorConfusion(String someValue) {
this.someValue = someValue;
}

public ConstructorConfusion(int someValue) {
this.someValue = "Number: " + Integer.toString(someValue);
}

...
}
```

then the following <constructor-arg> tag would be flagged because the class has more than one constructor

```
<bean id="constructorConfusion"
class="com.test.spring.ConstructorConfusion">
<constructor-arg>
<value>90</value>
</constructor-arg>
</bean>
```

**Use ApplicationContext to Assemble Beans**

**Severity:** Medium

**Summary**

Use the ApplicationContext, rather than imports, to assemble beans.

**Description**

This audit rule looks for places where imports are used to assemble beans. Like imports in Ant scripts, Spring import elements are useful for assembling modularized bean definitions. However, instead of pre-assembling them in the XML configurations using imports, it is more flexible to configure them through the ApplicationContext. Using the ApplicationContext also makes the XML configurations easier to manage.

**Example**

Given the following bean declaration:

```
<beans>
<import resource="billingServices.xml"/>
<import resource="orderServices.xml"/>
<import resource="shippingServices.xml"/>

<bean id="orderService"
class="com.test.spring.OrderService"/>
<beans>
```

the import lines would be flagged.

You can pass an array of bean definitions to the ApplicationContext constructor as follows:

```
String[] serviceResources = { "orderServices.xml",
"billingServices.xml", "shippingServices.xml" };
ApplicationContext orderServiceContext = new ClassPathXmlApplicationContext(
serviceResources);
```

**Use arraycopy() Rather Than a Loop**

**Severity:** Medium

**Summary**

The method arraycopy() should be used to copy arrays.

**Description**

This audit rule looks for places where a loop is being used to copy the elements of one array to another array. The method System.arraycopy() is much faster at copying array elements, so it should always be used when possible.

**Example**

The following loop would be flagged because it is only copying the elements of one array to another without performing any computation based on those elements:

```
employees = new Employee[people.length];
for (int i = 0; i < people.length; i++) {
employees[i] = (Employee) people[i];
}
```

**Use Available Constants**

**Severity:** Medium

**Summary**

Use available constants instead of creating new instances.

**Description**

Some classes provide constants for commonly used values. When possible, use these constants rather than creating new instances.

**Example**

The following test case would be flagged as a violation because `BigInteger.ZERO` could be used instead:

```
new BigInteger("0");
```

**Use Buffered IO**

**Severity:** Medium

**Summary**

All input and output should be buffered.

**Description**

This rule finds places where non-buffered IO classes are created and are not subsequently wrapped inside a buffered form of the class.

**Example**

The following FileReader creation would be flagged because it is not wrapped inside an instance of BufferedReader:

```
public FileReader getReader(String fileName)
{
```

```
return new FileReader(fileName);
}
```

It could be fixed by rewriting the code as follows:

```
public Reader getReader(String fileName)
{
return new BufferedReader(new FileReader(fileName));
}
```

**Use Camel Case in Namespaces**

**Severity:** Medium

**Summary**

Namespaces must not have spaces or underscores. Use hyphens or camel case for separation of words.

**Description**

This audit rule looks for namespace declarations, both WSDL and schema, that uses spaces or underscores in a namespace name.

**Example**

The following namespace uses underscores in a namespace and would thus be marked as a violation:

```
http://wsdl.example.com/wsdl/product/product_service/v1
```

**Use char Rather Than String**

**Severity:** Medium

**Summary**

Some String literals can be replaced by a character literal to improve performance when being used as parameter to some methods.

**Description**

This audit rule finds single character string literals as an argument to a method invocation where that argument can be replaced by a character literal to improve performance.

**Example**

Given the following declaration:

```
String s = "hello world";
```

The expression

```
s.indexOf("d")
```

would be flagged as needing to be replaced by the expression

```
s.indexOf('d');
```

**Use charAt() Rather Than startsWith()**

**Severity:** Medium

**Summary**

Use charAt() rather than startsWith() when the constant is a single character string.

**Description**

Use charAt(0) rather than startsWith("string constant") when the constant is a single character string. Using startsWith() with a one character argument works, but it makes several computations while preparing to compare its prefix with another string, which is unnecessary when you just want to compare one character against another.

**Example**

The following method invocation would be flagged as a violation:

```
string.startsWith("<")
```

because the condition could more efficiently be tested using:

```
string.length() > 0 && string.charAt(0) == '<'
```

**Use Compound Assignment**

**Severity:** Medium

**Summary**

Compound assignment statements are more compact and hence make the code easier to read.

**Description**

This audit rule looks for simple assignment statements that could be converted into compound assignment statements.

**Example**

The following assignment would be flagged as a violation:

```
sum = sum + array[i];
```

because it could be more compactly written as:

```
sum += array[i];
```

**Use deep Arrays methods when necessary**

**Severity:** Medium

**Summary**

Use deep `Arrays` methods when necessary.

**Description**

The `toString()`, `equals()`, and `hashCode()` methods in the Arrays class are shallow methods. That is, if you pass in an array of arrays, the elements in each sub-array will not be considered during the operation. Instead use the `deepToString()`, `deepEquals()`, and `hashCode()`

**Example**

The following would be flagged as a violation:

```
Arrays.toString(new int[][] {{1, 2, 3}, {4, 5, 6}});
```

**Use Domain-specific Terminology**

**Severity:** Medium

**Summary**

Use terminology applicable to the domain.

**Description**

Avoid using generic terms in identifiers. Use terms appropriate to the domain instead. This rule maps generic terms to specific terms according to a user-supplied list.

**Example**

If the term "thingy" were added as a generic term for the domain-specific term "widget", the following class name would be flagged as needing to be replaced by "MyWidget":

```
public class MyThingy
{
}
```

**Use equals() Rather Than ==**

**Severity:** Medium

**Summary**

Values should not be compared using equals (==) or not equals (!=).

**Description**

This audit rule finds places where two values are compared using either the equals (==) or not equals (!=) operators.

**Example**

The following expression would be flagged as a violation:

```
if (firstString == secondString) {
```

because it should be rewritten as:

```
if (firstString.equals(secondString)) {
```

**Use equals() Rather Than equalsIgnoreCase()**

**Severity:** Medium

**Summary**

Use String.equals() rather than String.equalsIgnoreCase() to compare strings.

**Description**

The method String.equals() should be used rather than the method String.equalsIgnoreCase() to compare strings.

**Example**

The following use of the method equalsIgnoreCase() would be flagged:

```
if (command.equalsIgnoreCase("delete")) {
```

**Use Existing Global Forwards**

**Severity:** Medium

**Summary**

Use the global forwards defined in the configuration file.

**Description**

This audit rule looks for ActionForwarder instances that refer to a page defined in configuration file. Such instances should return a global redirect to the referenced page in order to make it easier to modify the redirect.

**Example**

If the struts-config.xml file includes

```
<global-forwards>
<forward name="globalForwardName" path="/jsp/example.jsp" />
</global-forwards>
```

then the following execute method would be flagged:

```
public ActionForward execute(...)
throws Exception {
...
ActionForward forward = new ActionForward("/jsp/example.jsp");
return forward;
}
```

**Use ForwardAction Where Possible**

**Severity:** Medium

**Summary**

Do not create action classes that can be replaced with ForwardAction.

**Description**

This audit rule looks for actions that can be replaced with ForwardAction. Actions should be implemented using standard classes when possible in order to increase the flexability of the system.

**Example**

The following class would be flagged because it could be replaced by an instance of ForwardAction.

```
public class MyAction extends Action {
public ActionForward execute(ActionMapping mapping,
ActionForm form,
HttpServletRequest request,
HttpServletResponse response)
throws Exception {
return(mapping.findForward("some_forward_action_name"));
}
}
```

**Use html:messages Instead of html:errors**

**Severity:** Medium

**Summary**

Use html:messages instead of html:errors.

**Description**

This audit rule looks for uses of the html:errors tag within JSP files. When error messages need to be displayed to the end user, the html:messages tag should be used instead because it gets the markup language out of the resource bundle.

**Example**

The following JSP tag would be flagged:

```
<html:errors ... >
```

**Use idref Element**

**Severity:** Medium

**Summary**

Use the idref element to pass the id of another bean in the container.

**Description**

The idref element is an error-proof way to pass the id of another bean in the container (to a <constructor-arg/> or <property/> element):

```
<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">
<property name="targetName">
<idref bean="theTargetBean"/>
</property>
</bean>
```

The above bean definition snippet is exactly equivalent (at runtime) to the following snippet:

```
<bean id="theTargetBean" class="..."/>
```

```
<bean id="client" class="...">
<property name="targetName">
<value>theTargetBean</value>
</property>
</bean>
```

The main reason the first form is preferable to the second is that using the idref tag allows the container to validate at deployment time that the referenced, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the 'targetName' property of the 'client' bean. Any typo will only be discovered (with most likely fatal results) when the 'client' bean is actually instantiated. If the 'client' bean is a prototype bean, this typo (and the resulting exception) might only be discovered long after the container is actually deployed.

**Example**

The <value> tag in the following bean declaration would be flagged:

```
<bean id="client" class="...">
<property name="targetName">
<value>theTargetBean</value>
</property>
</bean>
```

**Use Interceptor to Track Duplicate Submits**

**Severity:** Medium

**Summary**

Don't use any token code in an Action class at all. The interceptors are designed to do all the work.

**Description**

This audit rule looks for calls to saveToken(), resetToken() and isTokenValid(). Advises to replace them with a corresponding interceptor.

**Example**

The following method would be flagged because of the invocation of the saveToken method:

```
public ActionForward execute(...) throws Exception {
...
// Create token.
```

```
saveToken(request);
return mapping.findForward("success");
}
```

Similarly, the following method would be flagged because of the invocations of the isValidToken and resetToken methods:

```
public ActionForward execute(...) throws Exception {
// Validate token for duplication submission.
if (!isTokenValid(request)) {
return mapping.findForward("duplicate");
} else {
...
}
// Reset token after transaction success.
resetToken(request);
return mapping.findForward("success");
}
```

**Use Interfaces for Collection Attributes**

**Severity:** High

**Summary**

Hibernate requires that collection-valued properties be typed to an interface such as java.util.Set or java.util.List and not to an actual implementation such as java.util.HashSet (this is a good practice).

**Description**

Hibernate requires interfaces for collection-typed attributes. You must use java.util.Set rather than HashSet, for example. At runtime, Hibernate wraps the HashSet instance with an instance of one of Hibernate's own classes. (This special class isn't visible to the application code). It is good practice to program to collection interfaces, rather than concrete implementations.

**Example**

childCategories would be flagged:

```
public class Category {
private String name;
private Category parentCategory;
private HashSet childCategories = new HashSet();

public Category() { }
```

```
...
}
```

**Use Interfaces Only to Define Types**

**Severity:** Medium

**Summary**

Use interfaces to define types, not as places to store constants.

**Description**

This rule identifies interfaces that do not define any methods. Interfaces that do not define any fields are ignored.

**Use locale-specific methods**

**Severity:** Medium

**Summary**

Locale-specific methods should be used to convert Strings to either upper or lower case.

**Description**

This rule looks for invocations of either toUpperCase() or toLowerCase() that do not have an instance of Locale as an argument. When converting a String to either upper or lower case, the locale-specific method should be used to ensure proper conversion.

**Example**

The following would be flagged as an error because the invocation of toLowerCase does not have a Locale as an argument:

```
if (x.toLowerCase().equals("test"))
```

**Use NoSuchElementException in next()**

**Severity:** Medium

**Summary**

An `Iterator`'s `next()` method should always be able to throw NoSuchElementExceptions in case a client ignores a false return from `hasNext()`.

**Description**

This rule looks for `Iterator`s whose `next()` method will never throw a NoSuchElementException.

**Example**

The following would be flagged as an error:

```
public class FooIterator implements Iterator {
public Object next(){};
}
```

**Use of "instanceof" with "this"**

**Severity:** Medium

**Summary**

The type of "this" should not be tested using the "instanceof" operator.

**Description**

This audit rule checks for places where the type of "this" is being tested using the "instanceof" operator. Code that depends on the type of an object should be distributed to the sublasses so that polymorphism will automatically choose the right code to execute.

**Example**

```
if (this instanceof SpecialSublass) ...
```

**Use Of Broken Or Risky Cryptographic Algorithm**

**Severity:** Medium

**Summary**

You should not use old or broken cryptography algorithms.

**Description**

This audit rule looks for uses of old or broken algorithms such as `MD4`, `MD5`, `SHA1`, `DES`.

**Security Implications**

Some older algorithms, once thought to require a billion years of computing time to break, can now be broken in days or hours. This includes `MD4`, `MD5`, `SHA1`, `DES`, and other algorithms which were once regarded as strong. Periodically check that your algorythm isn't obsolete.

**Example**

The following code will be flagged as a violation because the `MD5` algorithm is used:

```
public void func(...)
{
....
Cipher cipher = Cipher.getInstance("MD5");
....
}
```

**Use of instanceof in Catch Block**

**Severity:** Medium

**Summary**

Do not use `instanceof` to determine an exception's type in a catch block. Such check could miss some unexpected exception.

**Description**

This audit rule looks for `catch` blocks where the `instanceof` operator is used to check the exception's type. Subclasses of the caught exception can be handled separately by including a separate `catch` block for them before the superclass' `catch` block.

**Security Implications**

An uncaught exception can be handled by default exception handling mechanisms, which usually results in an exposure of a stack trace. This provides an attacker with an information on the technology stack of a system which could later be used to implement an attack.

**Example**

The following code will be flagged as a violation because the `instanceof` operator is used to check an exception's type:

```
} catch (IOException e) {
if (e instanceof EOFException) {
...
}
...
}
```

**Use of Random**

**Severity:** Medium

**Summary**

The class java.util.Random is not as secure as java.security.SecureRandom.

**Description**

This audit rule looks for any use of the class java.util.Random, including any classes declared as a subclass of Random, and any instances of Random being instantiated. The class java.security.SecureRandom should be used instead.

**Security Implications**

By using a cryptographically strong random number generated by SecureRandom, any risks that may be caused by a malicious user being able to anticipate the outcome of a random number, will be prevented.

**Example**

The following would be flagged as a violation:

```
class A extends Random {}
```

**Use of xs:any**

**Severity:** Medium

**Summary**

Do not use `<xs:any.../>` in your schemas.

**Description**

This audit rule looks for any usage of `<xs:any>`. Using `<xs:any>` causes validation complications, and can require excessive code to ensure proper values are passed in schemas.

**Example**

The following element declaration uses `<xs:any>` and would thus be marked as a violation:

```
<xs:element name="person">
<xs:complexType>
<xs:sequence>
<xs:element name="firstname" type="xs:string"/>
<xs:element name="lastname" type="xs:string"/>
<xs:any minOccurs="0"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

**Use Only Non-Final Persistent Classes**

**Severity:** Low

**Summary**

Use only non-final persistent classes.

**Description**

This audit rule looks for persistent classes for which a proxy cannot be created. The use of proxies, a central feature of Hibernate, depends on the persistent class being either non-final, or the implementation of an interface that declares all of the public methods.

You can persist final classes that do not implement an interface with Hibernate, but you won't be able to use proxies for lazy association fetching - which will limit your options for performance tuning.

**Example**

The following persistent class would be flagged because it is final.

```
public final class Message {
...
}
```

**Use Only Static Inner Classes**

**Severity:** Medium

**Summary**

Use only static inner persistent classes.

**Description**

This audit rule looks for inner classes that are declared to be persistent and checks to make sure that they are static. Hibernate cannot persist non-static inner classes.

**Example**

If the mapping file OuterClass$NestedClass.hbm.xml contains

```
<class
name="OuterClass$NestedClass" ... >
```

then the persistent class NestedClass would be flagged because it non-static:

```
public final class OuterClass {
...

public class NestedClass {
...
}
}
```

**Use Or to Combine SWT Style Bits**

**Severity:** Medium

**Summary**

SWT style bits should be combined using the bitwise-OR operator.

**Description**

This audit rule looks for creations of SWT widgets where the style bits are being combined using an operator other than the bitwise-OR.

**Example**

```
Text text = new Text(shell, SWT.SINGLE & SWT.BORDER);
```

**Use Privileged Code Sparingly**

**Severity:** Medium

**Summary**

Prevents the use or overuse of privileged code.

**Description**

This audit rule flags instances of java.security.PrivilegedAction and java.security.PrivilegedExceptionAction which have more than a specified number (default 0) of statements within run().

Note: When set to default value of 0, no privileged code will be allowed since the method run, see PrivilegedAction or PrivilegedExceptionAction, returns an Object, which at minimum takes one statement: "return null;".

**Security Implications**

Privileged code allows code access to system variables the Java API would normally not allow access to, for security purposes privileged code should be used sparingly.

**Example**

The following source would be flagged since there are statements in a run() method declared for a PrivilegedAction.

```
AccessController.doPrivileged(new PrivilegedAction() {
public Object run() {
System.loadLibrary("awt");
return null;
}
});
```

**Use Qualified Attributes**

**Severity:** Medium

**Summary**

The elementFormDefault and attributeFormDefault schema attributes must both be set to `qualified`.

**Description**

This audit rule looks for schema declarations that do not require elementFormDefault and attributeFormDefault schema attributes to be required. Not doing so means that instance documents will have all elements prefixed with the namespace abbreviations. While this will result in larger instance documents, it will be easier to debug and trouble shoot issues with instance documents. Avoid using excessively long prefix's in large documents.

**Example**

The following schema declaration does not require elementFormDefault and attributeFormDefault schema attributes to be required and would thus be marked as a violation:

```
<schema
targetNamespace="http://faulty-domain.com/stockquote.xsd"
xmlns="http://www.w3.org/2000/10/XMLSchema">
...
</schema>
```

**Use Session-per-request Pattern**

**Severity:** Medium

**Summary**

Don't use the session-per-operation antipattern.

**Description**

This audit rule looks for uses of the session-per-operation antipattern in which a session is opened and closed for every simple database call in a single method, which in turn required beginning and committing a database transactions for each database call as well.

Database calls in an application are made using a planned sequence, they are grouped into atomic units of work. In the session-per-request model, a request from the client is send to the server (where the Hibernate persistence layer runs), a new Hibernate Session is opened, and all database operations are executed in this unit of work. Once the work has been completed (and the response for the client has been prepared), the session is flushed and closed. You would also use a single database transaction to serve the clients request, starting and committing it when you open and close the Session.

This rule assumes that each method that creates a session is performing a single unit of work. If there are methods performing multiple units of work, they should be refactored into separate methods.

**Example**

The following method would be flagged because it use the session-per-operation antipattern.

```
private void addPersonToEvent(Long personId, Long eventId) {
...
Session session1 = sessionFactory.getCurrentSession();
session.beginTransaction();
Person aPerson = (Person) session.load(Person.class, personId);
session.getTransaction().commit();

Session session2 = sessionFactory.getCurrentSession();
session.beginTransaction();
Event anEvent = (Event) session.load(Event.class, eventId);
session.getTransaction().commit();

Session session3 = sessionFactory.getCurrentSession();
session.beginTransaction();
aPerson.getEvents().add(anEvent);
session.getTransaction().commit();
...
}
```

It should be replaced by something like the following

```
private void addPersonToEvent(Long personId, Long eventId) {
...
```

```
Session session = sessionFactory.getCurrentSession();
session.beginTransaction();

Person aPerson = (Person) session.load(Person.class, personId);
Event anEvent = (Event) session.load(Event.class, eventId);
aPerson.getEvents().add(anEvent);

session.getTransaction().commit();
...
}
```

**Use Setter Injection**

**Severity:** Medium

**Summary**

Use setter injection rather than constructor injection.

**Description**

This audit rule looks for beans that use constructor injection when they could be using setter injection. Spring provides three types of dependency injection: constructor injection, setter injection, and method injection. Typically we only use the first two types. Constructor injection can ensure that a bean cannot be constructed in an invalid state, but setter injection is more flexible and manageable, especially when the class has multiple properties and some of them are optional.

**Example**

The following bean declaration would be flagged because it uses constructor injection:

```
<bean id="orderService"
class="com.test.spring.OrderService">
<constructor-arg ref="orderDAO"/>
</bean>
```

it should be changed to:

```
<bean id="orderService"
class="com.test.spring.OrderService">
<property name="orderDAO" ref="orderDAO">
</bean>
```

**Use Start Rather Than Run**

**Severity:** Medium

**Summary**

Threads should be started rather than run.

**Description**

This audit rule looks for places where a thread is run using the run() method, rather than started using the start() method. This is usually a mistake since it causes the run() method to be run in the calling thread rather than in a newly spawned thread.

**Example**

The following invocation would be flagged:

```
thread.run();
```

**Use StringBuffer length()**

**Severity:** Medium

**Summary**

Test the length of a StringBuffer or StringBuilder using its length() method directly rather than converting it to a string first.

**Description**

This audit rule looks for places where the length of a StringBuffer or StringBuilder is computed by first converting it to a String.

**Example**

The following would be flagged:

```
StringBuffer sb = new StringBuffer("Foo");
if (sb.toString().length() == 0) ...
```

**Use StrutsTestCase for Unit Testing**

**Severity:** Medium

**Summary**

Use StrutsTestCase instead of TestCase for unit-testing Struts classes.

**Description**

This audit rule looks for subcasses of TestCase that include methods to test Struts-related classes. They should subclass StrutsTestCase in order to make use of the additional functionality defined by it.

**Example**

The following class would be flagged because it extends TestCase even though it is clearly testing a Struts class.

```
public class LoginActionTest extends TestCase {
public void testExecute() {
LoginAction loginAction = new LoginAction();
assertEquals(...);
...
}
}
```

**Use SuccessAction Where Possible**

**Severity:** Medium

**Summary**

Do not create action classes that can be replaced with SuccessAction.

**Description**

This audit rule looks for actions that can be replaced with SuccessAction. Actions should be implemented using standard classes when possible in order to increase the flexability of the system.

**Example**

The following class would be flagged because it could be replaced by an instance of SuccessAction.

```
public class MyAction extends Action {
public ActionForward execute(ActionMapping mapping,
ActionForm form,
HttpServletRequest request,
HttpServletResponse response)
throws Exception {
return mapping.findForward("success");
}
}
```

**Use Thread-safe Lazy Initialization**

**Severity:** Medium

**Summary**

Static fields should be initialized in a thread safe way.

**Description**

Static fields are typically initialized either as part of their declaration, in a static initializer, or lazily in a static method. The first two ways are thread safe because of the way the JVM initializes classes. In order for the initialization of a lazily initialized field (such as the unique instance of a Singleton class) to be thread safe, either the method must be synchronized or the body of the method must be inside a synchronize statement. If not, and if the method is called by multiple threads, one of the threads might get a reference to the field before it is fully initialized or it might be initialized multiple times. This audit rule looks for places where a static field is initialized in a way that is not thread safe.

**Example**

The following singleton is not thread-safe, and would be marked as a violation.

```
public class MySingleton {
public MySingleton instance = null;

public MySingleton getInstance() {
if (instance == null) {
instance = new MySingleton();
}
return instance;
}
```

```
private MySingleton() {}
}
```

## Use Type for Constructor Argument Matching

**Severity:** Medium

**Summary**

Use the type attribute, rather than the index attribute, for constructor argument matching.

**Description**

Spring allows you to use a zero-based index to solve the ambiguity problem when a constructor has more than one arguments of the same type, or when value tags are used. Using an index is somewhat less verbose, but it is more error-prone and hard to read compared to using the type attribute. You should only use the index attribute when there is an ambiguity problem in the constructor arguments.

**Example**

For example, instead of using the index attribute as in the following:

```
<bean name="billingService" class="example.test.BillingService">
<constructor-arg index="0">
<value>90</value>
</constructor-arg>
</bean>
```

it is better to use the type attribute like this:

```
<bean name="billingService" class="example.test.BillingService">
<constructor-arg type="int">
<value>90</value>
</constructor-arg>
</bean>
```

## Use Type-Specific Names

**Severity:** Medium

**Summary**

Variables of some types should have a well-known name.

**Description**

This audit rule checks for variable declarations where the type of the variable is a known type but whose name is not one of the names approved for variables of that type.

**Example**

If the class java.io.InputStream were on the list with the names "in" and "input" as the only valid names, the following declaration would be flagged as a violation:

```
InputStream stream;
```

## Use Valid SWT Styles

**Severity:** High

**Summary**

Widget types should only be assigned valid SWT styles.

**Description**

This audit rule looks for declarations of widget types with incorrect styles set. Incorrectly set styles create a user interface not intended by the developer, are unnecessary, and can lead to program errors or exceptions. Styles input into a constructor are assumed to directly reference the integer values, that is, local integer fields or integer constants are assumed not to be passed instead.

Under the audit rule parameters ("Parameters" tab) the sets of valid styles for each widget can be changed. After selecting a widget from the left-hand list, say Label, the list on the right is the set of "incompatible style groups". For the Label widget, you will see both "SWT.SEPARATOR" and "SWT.SHADOW_IN | SWT.SHADOW_OUT | SWT.SHADOW_NONE" in this list. This means that these four styles are all valid styles for the Label widget.

Furthermore, since SWT.SHADOW_IN, SWT.SHADOW_OUT and SWT.SHADOW_NONE are in a group, only one of these styles may be attributed to a Label. If two of these three are given to a Label, the constructor will be flagged.

The buttons 2.0/.../3.2 restore all widgets and styles included in the SWT package for the specified version. All non-SWT widgets entered by the user, aren't affected by these buttons.

**Example**

The following would be flagged twice as Button widgets can't have the

SWT.SINGLE style and can also not contain both the SWT.CHECK and SWT.PUSH styles at the same time.

```
Button button = new Button(shell, SWT.SINGLE | SWT.CHECK | SWT.PUSH);
```

## Use Valid Type for Form-property

**Severity:** Medium

**Summary**

All form-property tags in the Struts configuration file should have a valid type property.

**Description**

This audit rule looks for form-property tags in the Struts configuration file that have an invalid type property. The type property should have one of the following values:

* java.math.BigDecimal
* java.math.BigInteger
* boolean and java.lang.Boolean
* byte and java.lang.Byte
* char and java.lang.Character
* java.lang.Class
* double and java.lang.Double
* float and java.lang.Float
* int and java.lang.Integer
* long and java.lang.Long
* short and java.lang.Short
* java.lang.String
* java.sql.Date
* java.sql.Time
* java.sql.Timestamp

You may also specify Arrays of these types (e.g. String[]) as well as concrete implementations of the Map interface, such as java.util.HashMap, or a List implementation, such as java.util.ArrayList.

**Example:**

The properties "page" and "data" would be flagged because they are using invalid types.

```
<form-bean name="checkoutForm"
type="org.apache.struts.validator.DynaValidatorForm">
<form-property name="address" type="java.lang.String"/>
<form-property name="age" type="java.lang.Integer"/>
<form-property name="page" type="java.lang.StringBuffer"/>
<form-property name="data" type="java.lang.Object"/>
</form-bean>
```

## Use valueOf() to wrap primitives

**Severity:** Medium

**Summary**

When wrapping primitives, always use the valueOf() method to convert them instead of calling the constructor.

**Description**

The valueOf() methods in the wrapper classes cache commonly used values. It is therefore more efficient to use them, rather than to use the constructor which creates a new instance every single time.

**Example**

The following would be marked as a violation:

```
new Integer(5);
```

## Use WSDL First approach

**Severity:** Medium

**Summary**

All services must do WSDL First development.

**Description**

This audit rule looks for names that indicate that an automatic WSDL generation approach was used. Specifically, it looks for the usage of "parameters" as a parameter (wsdl:part) name. The only exception is Application Services that are wrapping components. Application services that are wrapping existing components have the option to use or not use WSDL First. WSDL First development requires you to manually create the WSDL before creating the Java code.

**Example**

The following message declaration contains a parameter with a meaningless name and would thus be marked as a violation:

```
<wsdl:message name="rpmDateLookupRequest">
<twsdl:part
```

```
element="intf:rpmDateLookup"
name="parameters"/>
<wsdl:message/>
```

## Use XmlBeanFactory

**Severity:** Medium

**Summary**

Use the XML configuration format of BeanDefinition information for all but the most trivial of applications.

**Description**

Spring provides two main BeanFactory implementations. The first, DefaultListableBeanFactory, reads the BeanDefinition information from a property file using the PropertiesBeanDefinitionReader.

The second, XmlBeanFactory, allows you to manage your bean configuration using XML rather than properties. Although properties are ideal for small, simple applications, they can quickly become cumbersome when you are dealing with a large number of beans. For this reason, it is preferable to use the XML configuration format for all but the most trivial of applications. The XmlBeanFactory is derived from DefaultListableBeanFactory and simply extends it to perform automatic configuration using the XmlBeanDefinitionReader.

**Example**

The following main method would be flagged because of its use of DefaultListableBeanFactory:

```
public class PropertiesConfig {

public static void main(String[] args) throws Exception {
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(
factory);

Properties props = new Properties();
props.load(new FileInputStream("..."));
rdr.registerBeanDefinitions(props);

...
}
}
```

you should do this instead:

```
public class XmlConfig {
```

```
public static void main(String[] args) throws Exception {
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("..."));

...
}
}
```

## Validate XML

**Severity:** Medium

**Summary**

All XML data entering the program should be validated by a parser against a DTD or XML Schema.

**Description**

Validating incoming data against a predefined model leaves less space for errors that could be exploited by an attacker. All XML data entering the program should be validated either against a DTD (using factory's setValidating(true) method), or against an XML Scheme (using factory's setSchema(Schema schema) method), or using Java 5 javax.xml.validation facilities.

**Security Implications**

Exploits in the data model can be used by an attacker to perform an injection attack.

**Example**

The following code does not use any of validation techniques before accessing and using DOM data and would thus be marked as violation:

```
DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(url).getDocum
entElement();
```

## Validation Method Naming Convention

**Severity:** Medium

**Summary**

Verifies that the first component of a backing bean's validation method's name is "validate".

**Description**

By a common convention, the names of validation methods in backing beans start with word "validate". This audit rule looks for violations of this convention.

**Example**

The following method will be flagged because it's name does not start with "validate".

```
public class MyBackingBean {
public void emailValidator(FacesContext context, UIComponent comp, Object
value) {
}
}
```

## Validation Not Enabled

**Severity:** Medium

**Summary**

Validation plug-in is disabled by default and should be explicitly enabled in Struts configuration.

**Description**

This rule violates struts-config.xml configuration file that does not explicitly specify Validation framework enabled.

**Security Implications**

Unvalidated input can be used by an attacker to perform all kinds of remote attacks.

**Example**

The following code declares the usage of Struts validation framework. If struts-config.xml configuration file does not contain this code, it will be violated:

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
<set-property property="pathnames" value="/technology/WEB-INF/validator-
rules.xml, /WEB-INF/validation.xml"/>
</plug-in>
```

## Validator Configuration File Does Not Exist

**Severity:** Medium

**Summary**

XML files referenced as validator configuration files in the struts-config.xml file should exist.

**Description**

The Struts configuration file may specify two validator configuration files. This audit rule checks to ensure that every declared validator configuration file exists.

**Example**

Given the following content in the struts-config.xml file:

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
<set-property property="pathnames" value="/WEB-INF/validator-rules.xml, /WEB-
INF/validation.xml"/>
</plug-in>
```

The appropriate value would be flagged, if either validator-rules.xml or validation.xml does not exist.

**Validator Disabled**

**Severity:** Low

**Summary**

You shouldn't disable the form's validate() method in the action form mapping.

**Description**

An action form mapping should never disable validation. Disabling validation disables the Struts Validator as well as any custom validation logic performed by the form.

**Example**

The following action form mapping would be flagged because it disables validation

```
<action path="/download"
type="com.test.action.DownloadAction"
name="downloadForm"
scope="request"
input=".download"
validate="false">
</action>
```

**Variable Declared Within a Loop**

**Severity:** Low

**Summary**

One should be careful when declaring variables inside of for, while, or do loops. This may lead to problems especially if the variable is initialized as well.

**Description**

This audit rule looks for variables that have been declared or initialized within a for, while, or do loop.

Declaring a variable within a loop breaks some suggested style rules.

Initializing a variable within a loop may indicate a problem with the program's logic. It could also lead to degraded performance.

**Example**

The following declaration of the variable "event" would be flagged as a violation:

```
while (hasMoreEvents()) {
Event event = getNextEvent();
...
}
```

**Variable Has Null Value**

**Severity:** Medium

**Summary**

A variable that is guaranteed to have a null value and is used in an expression may indicate that the programmer forgot to initialize variable with its actual value.

**Description**

This rule looks for a places where variables with `null` values are used in an expression.

**Security Implications**

Such an error may indicate a flaw in the program's logic that may leave the software vulnerable if present in the security-sensitive part of an application.

**Example**

The following usage of variable should be marked as violation because the variable is always `null`:

```
public boolean myMethod(String param)
{
String tmp = null;
if (tmp.equals(param)) {
return true;
} else {
return false;
}
}
```

**Variable Should Be Final**

**Severity:** Medium

**Summary**

Variables that are assigned only once should be final. Variables marked as final communicate additional information to the reader about how the variable is supposed to be used.

**Description**

This audit rule finds

* private fields that have a value assigned in an initializer or in a constructor and no where else,

* method parameters that are not assigned a value in the method,

* catch parameters that are not assigned a value in the catch block, and

* local variables that have a value assigned in their initializer and no where else.

The variable should be marked as final indicating that the value of the variable does not change through out the variable's lifetime.

**Security Implications**

When a field is only assigned a value in one place, this usually indicates that the field is not supposed to be changed during the life cycle of the object. By marking the field as `final` you reduce the risk that a malicious user can alter its value and thus create an unexpected situation with unpredictable results.

**Variable Usage**

**Severity:** Medium

**Summary**

Variables should never shadow variables with the same name that are defined in an outer scope.

**Description**

This audit rule checks for any declarations of variables that shadow a variable with the same name that is defined in an outer scope.

**Example**

In a class with a field declaration such as:

```
private int inventoryCount;
```

the following parameter would be flagged as a violation:

```
public void printInventory(int inventoryCount)
```

**Wait Inside While**

**Severity:** Medium

**Summary**

The wait() method should only be invoked within a while loop.

**Description**

This audit rule looks for invocations of the wait() method that occur outside of a while loop.

**Example**

The following invocation of the wait() method would be flagged as a violation:

```
public void waitForEvent()
{
synchronize (eventQueue) {
eventQueue.wait();
...
}
}
```

**Wait method invoked while two locks are held**

**Severity:** Medium

**Summary**

Do not invoke the `wait` method while two locks are being held.

**Description**

This rule looks for places where the code invokes the `wait()` method while two locks are held.

**Security Implications**

Situation when some threads wait on monitor while two locks are held can lead to deadlock.

**Example**

The following invocation of the `wait` method will be flagged as a violation because two lock are being held.

```
public synchronized void myMethod(Object obj)
{
```

```
synchronize (obj) {
obj.wait();
}
}
```

### wait() Invoked Instead of await()

**Severity:** Medium

**Summary**

If class is subtype of `java.util.concurrent.locks.Condition` you should use await method instead of wait method.

**Description**

This audit rule looks for places where the `wait()` method is invoked on a `java.util.concurrent.locks.Condition` object.

**Security Implications**

Invoking the `wait()` method on a `java.util.concurrent.locks.Condition` object is usually a mistake. This class is specially used for synchronization and provides the method await(), similar to wait() in its name. Accidentially calling wait() instead is a plain error which will result in the application behaving in an unexpected way and possibly becoming vulnurable to an attack. Asynchronous environment is especially vulnurable to such errors as deadlock may occur and ultimately lead to denial-of-service state.

**Example**

The following code will be marked as a violation because the `wait()` method used:

```
public void myFunction( Condition cond )
{
try{
cond.wait();
catch( InterruptException e){
return;
}
}
```

### Weblogic Session ID Length

**Severity:** Medium

**Summary**

To protect a session from brute-force attacks, the session identifier length should be set to at least 24 bytes.

**Description**

Estimations of the time and resources required to brute-force a session based on its identifier length show that to provide sufficient protection the identifier should be at least 24 bytes long. See details of an estimation at CWE-6 J2EE Misconfiguration: Insufficient Session-ID Length common weakness description. Weblogic allows you to define the session id length in a configuration file. This rule finds Weblogic session id length declarations that set the session id length too short.

**Security Implications**

Session ID of insufficient length can be bruteforced, thus granting an attacker a full access to the user's session.

**Example**

The following code declares a session id length of 8 bytes and would thus be marked as violation:

```
<session-descriptor>
<session-param>
<param-name>
IDLength
</param-name>
<param-value>8</param-value>
</session-param>
</session-descriptor>
```

**White Space Before Property Name**

**Severity:** High

**Summary**

Property names should not be preceded by white space.

**Description**

This audit rule looks for property names that are preceded by white space. Property names that are preceded by white space often indicate a missing line continuation character on the preceding line.

**Example**

The property named "operation" below, which was likely intended to be part of the line before it, would be flagged as a violation:

```
messageText = You should have known better than to have tried this
operation without first changing your preference settings.
```

**White Space Usage**

**Severity:** Low

**Summary**

Indentation should consistently be done using either tabs or spaces.

**Description**

Indentation should either be done using tabs, or it should be done using spaces, but should not be done using a combination of the two. This audit rule allows you to specify the indentation style you prefer, and then looks for places where space and tab characters are used in ways that violate the specified criteria.

**Wrong Family Returned**

**Severity:** Medium

**Summary**

The method getFamily should return family declared in the configuration file.

**Description**

This audit rule looks for subclasses of UIComponentBase that implement a getFamily method that returns a constant value that does not match the one from faces-config.xml. The value returned from the getFamily() method must match the value specified for the component in the application configuration file.

**Example**

A violation will be signalled if "Checkbox" is not the component family specified for "RoundButton" component in the configuration file.

```
public class RoundButton extends UIComponentTag {
...
public String getFamily() {
return "Checkbox";
}
...
}
```

**Wrong Integer Type Suffix**

**Severity:** Low

**Summary**

Long literals should use 'L' for a suffix.

**Description**

This audit rule looks for long-valued literals whose suffix is a lower-case 'l'. Although the language specification allows this, it is too easily confused with the number one (1), and hence should not be used.

**Example**

```
public static final long ONE = 1l;
```

**WSDL Must Specify ESB Endpoints**

**Severity:** Medium

**Summary**

A WSDL must specify ESB endpoints. The soap location must contain "ccx/cc-router".

**Description**

This audit rule looks for declarations of `wsdlsoap:address` without either a `location` attribute defined or without a `ccx/cc-router` portion of URI in its value.

**Example**

The following declaration of `wsdlsoap:address` doesn't use `ccx/cc-router` in its `location` attribute definition:

```
<wsdlsoap:address
location="http://<tier>esb/ServiceName/version"/>
```

**WSDL Namespace for Included Schemas**

**Severity:** Medium

**Summary**

Message schemas can be included in WSDL documents. If they are included in the WSDL, the WSDL namespace can be used.

**Description**

This audit rule looks for message schemas that use a WSDL namespace without being included in a WSDL document.

**Example**

The following definition of a message will be marked as a violation because it isn't included in the WSDL document:

```
<wsdl:message name="messageName">
....
</wsdl:message>
```

**XSD File Naming Convention**

**Severity:** Medium

**Summary**

XSD file names must conform to a standard.

**Description**

This audit rule looks for XSD files whose names do not contain one of the listed keywords or that include a version number (defined as a list of numbers separated with dots).

**Example**

The following file names would be flagged as violations, the first because it does not include a listed keyword and the second because it includes a version number:
SystemData.xsd
Enterprise_SystemData_v1.2.xsd