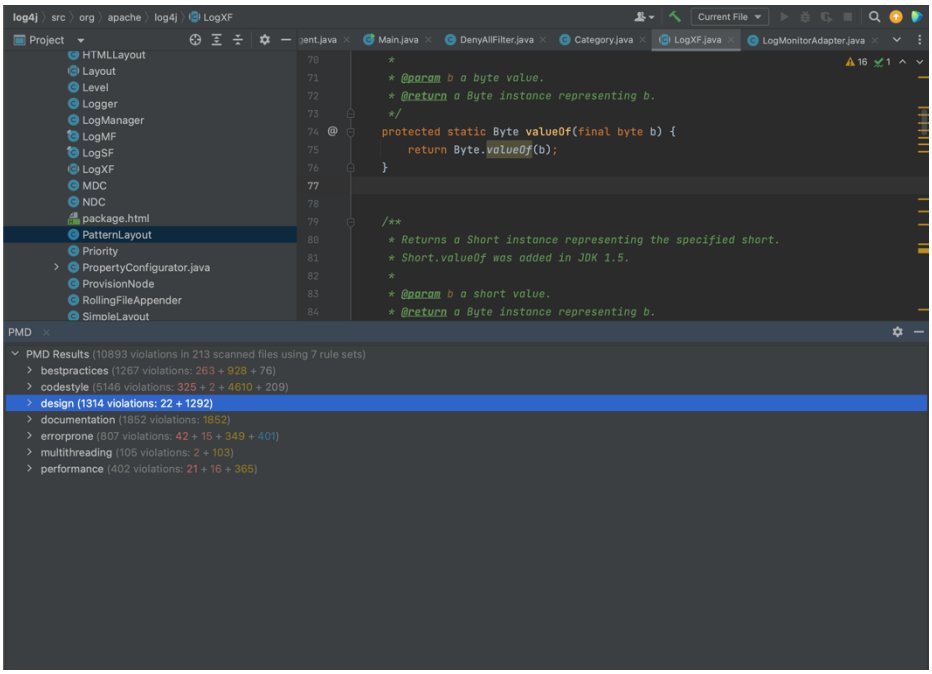
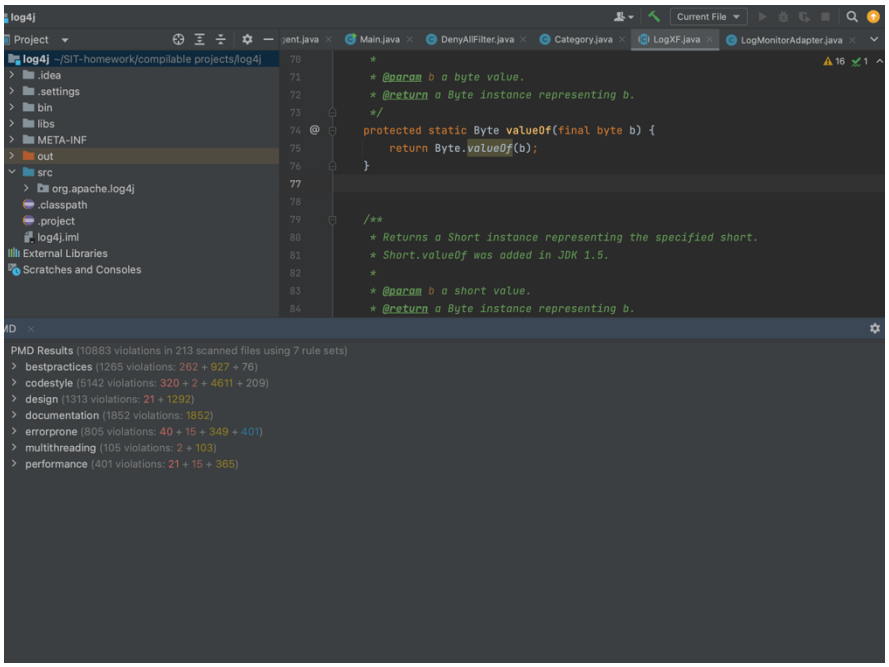
Bug fixes

Result:

Before bug fixed, PMD result report:



After 10 bugs fixed:



Honor pledge: I pledge my honor that I have abided by the Stevens Honor System.

-------Jiayin Huang 10477088

BestPractices:
Item1 AvoidReassingningParamers(16 violations)
True positive

a. PropertyPrinter.foundProperty()

```
public void foundProperty(Object obj, String prefix, String name,
Object value) {
  // XXX: Properties encode value.toString()
  if (obj instanceof Appender && "name".equals(name)) {
    return;
  }
  if (doCapitalize) {
    name = capitalize(name);
  }
  out.println(prefix + name + "=" + value.toString());
}
```

fix:

```
public void foundProperty(Object obj, String prefix, String name, Object value) {
   // XXX: Properties encode value.toString()
   if (obj instanceof Appender && "name".equals(name)) {
      return;
   }

   String updatedName = name;

   if (doCapitalize) {
      updatedName = capitalize(updatedName);
   }

   out.println(prefix + updatedName + "=" + value.toString());
}
```

Reason: By introducing a new local variable updatedName and using it instead of modifying the input parameter name, we are adhering to this best practice. This change ensures that the input parameter's original value remains unaltered within the method, making the code easier to understand and maintain. It also eliminates the PMD rule violation for AvoidReassigningParameters.

It took me one hour to fix this.

Item2:

GuardLogStatement(226 violations)
True positive

a. FileAppender.closeFile()

```
protected
 void closeFile() {
   if(this.qw != null) {
     try {
this.qw.close();
     }
     catch(java.io.IOException e) {
         if (e instanceof InterruptedIOException) {
             Thread.currentThread().interrupt();
         }
// Exceptionally, it does not make sense to delegate to an
// ErrorHandler. Since a closed appender is basically dead.
LogLog.error("Could not close " + qw, e);
     }
   }
 }
```

fix:
Reason:
To fix the GuardLogStatement violation in the closeFile method, we should guard the logging statement with a conditional that checks if logging is enabled for the desired level. In this case, the logging level is "error". Here's the modified code:

```
protected void closeFile() {
   if (this.qw != null) {
      try {
         this.qw.close();
      } catch (java.io.IOException e) {
         if (e instanceof InterruptedIOException) {
            Thread.currentThread().interrupt();
         }
         // Exceptionally, it does not make sense to delegate to an
         // ErrorHandler. Since a closed appender is basically dead.
         if (LogLog.isDebugEnabled()) { // Guard the logging statement
            LogLog.error("Could not close " + qw, e);
         }
      }
   }
}
```

By adding the if (LogLog.isDebugEnabled()) conditional, we ensure that the logging statement is only executed when the logging level is set to "debug" or lower. This avoids the GuardLogStatement violation and can also help improve performance by avoiding unnecessary logging when it's not needed.

It took me an hour to fix it.

Item 3:
SystemPrintln(21 violations)
True Positive

a. LogLog.debug()

```
public
static
void debug(String msg) {
  if(debugEnabled && !quietMode) {
    System.out.println(PREFIX+msg);
  }
}
```

Fix:
Reason:
To fix the SystemPrintln violation in the LogLog.debug() method, we should use a logging framework instead of directly calling System.out.println.

First, add an import for java.util.logging.Logger. Then, create a Logger instance as a class member. replace the System.out.println call with the LOGGER.log method:

```
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public static void debug(String msg) {
   if (debugEnabled && !quietMode) {
      LOGGER.log(Level.FINE, PREFIX + msg); // Use java.util.logging.Logger instead of
System.out.println
   }
}
```

By using java.util.logging.Logger and setting the log level to Level.FINE, we ensure that the logging statement is only executed when the logging level is set to "fine" or lower, and we avoid the SystemPrintln violation.

It took me an hour to fix it.

codestyle
Item4:

a. EmptyMethodInAbstractClassShouldBeAbsract(5 violations)
   True positive

```
public
void activateOptions() {
}
```

Fix it:
Reason:
To fix the EmptyMethodInAbstractClassShouldBeAbstract violation, we should declare the activateOptions method as abstract, assuming that this method is part of an abstract class.

public abstract void activateOptions();

By declaring the method as abstract, we indicate that it should be implemented by any concrete subclass of the abstract class.

It took me 30 mins to fix it.

Item5: FieldNamingConventions(148 violations)
True positive

a. HTMLLayout

```
protected final int BUF_SIZE = 256;
```

Fix it:
Reason:
Should follow Java's naming conventions for fields. In Java, fields should be in camelCase starting with a lowercase letter.

protected final int bufSize = 256;

It took me 20 minutes to fix it.

Item6: FinalParameterinAbstractMethod(5 violations)
Is a false

a. PatternConverter.format()

```
public abstract void format(final Object obj, final StringBuffer
toAppendTo);
```

Fix it:
Reason:
To fix it (assuming it is a true positive), remove the final keyword from the parameters:

public abstract void format(Object obj, StringBuffer toAppendTo);

The FinalParameterInAbstractMethod violation suggests that using final parameters in abstract methods may not be necessary because a subclass implementation may require flexibility in modifying the parameter.

It took me 30 minutes to fix it.

Item7: FormalParameterNamingConventions(9 violations)
DOMconfigurator.parseLayout()
Is a false

a.

```
Layout parseLayout (Element layout_element)
```

Fix it;
Reason:
Typically, formal parameters should be in camelCase and should not contain underscores.

In this case, the parameter layout_element does not follow the naming convention.

```
Layout parseLayout(Element layoutElement) {
  // The rest of the method implementation
}
```

It took me 30 minutes to fix it.

Iteam8:
Design:
AvoidThrowinngNullPointerException(3 violations)

a. PatternParser.parse()

```
public static void parse(
  final String pattern, final List patternConverters,
  final List formattingInfos, final Map converterRegistry, final Map
rules) {
  if (pattern == null) {
    throw new NullPointerException("pattern");
  }
```

Fix it:
Reason:

To fix the AvoidThrowingNullPointerException violation, we should replace the NullPointerException with a more specific exception, such as IllegalArgumentException. This exception is more appropriate for cases where an invalid argument is passed to a method.

```
public static void parse(
  final String pattern, final List patternConverters,
  final List formattingInfos, final Map converterRegistry, final Map rules) {
  if (pattern == null) {
    throw new IllegalArgumentException("Pattern parameter cannot be null.");
  }
```

It took me an hour to fix it.

Item9: errorprone(807 violations: 42 + 15 + 349 + 401)

DenyAllFilter.getOptionStrings()
True positive

a.

```
public
String[] getOptionStrings() {
  return null;
}
```

Fix it:
Reason:
To fix the error-prone violation in the DenyAllFilter.getOptionStrings() method, we should avoid returning null directly. Instead, we can return an empty array. This way, the callers of this method won't encounter unexpected NullPointerExceptions when they try to work with the returned array.

```
public String[] getOptionStrings() {
  return new String[0];
}
```

It took me 30 minutes to fix it.

Item10: performace(402 violations: 21 + 16 +365)

ByteInstantiation(1 violation)
Is a false

a.

```
protected static Byte valueOf(final byte b) {
    return new Byte(b);
}
```

Fix it:
Reason:
The ByteInstantiation violation occurs when a Byte object is created using the new Byte() constructor, which can be inefficient. In this case, it's better to use the Byte.valueOf() method, which uses a cache for better performance.

```
protected static Byte valueOf(final byte b) {
   return Byte.valueOf(b);
}
```

It took me 30 minutes to fix it.