Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate

Software quality evaluation via static analysis

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
luigi.lavazza@uninsubria.it

The 14th International Conference on Software Engineering Advances
ICSEA 2019
November 24, 2019 – Valencia, Spain

Motivations

- During software development, identifying and correcting defects as soon as possible is of paramount importance, because the longer a defect survives, the more expensive it is to remove it.
- In this tutorial we concentrate on code defects (aka bugs). To minimize the cost of detecting and correcting bugs, we must be able to identify bugs in the coding phase.
- We also would like to make the identification as quick and cheap as possible.
- To achieve these goals, tools performing static analysis of code can be used.

Motivations

- Static analysis tools are able to spot potential defects in code.
- Due to theoretical limitations, **they cannot indicate with certainty the existence of a bug**; hence their findings have to be verified by developers.
- Nonetheless, these tools can be very effective and efficient in spotting bugs.

Contents

- In this tutorial, we shall see how to use tools that analyze Java code, looking for both "general" bugs and security flaws.
- In additions, we shall have a look at measurement tools: these tools compute code measures, which can be used to guide manual inspections,
  - classes or methods that feature extreme values of measures uysually deserve to be analysed with greater attention

Automated static analysis

- Static analysis makes it possible to analyze software artifacts–namely code–without executing them.
- Static analysis con be performed
  - Manually by people (via inspections)
  - Automatically, via tools
- Of course, automated static analysis is possible only if the element to be analyzed is written in a formal, machine-understandable language
- Automated static code analysis is very effective
  - It can also be performed in addition to inspections

- Beware: it has limits due to the undecidability of several interesting properties.

Limits of automated static analysis

- Unfortunately, many programs that would be very useful cannot be written.
  - For instance, there is no program that, given two other programs P1 and P2, is able to decide if P1 and P2 are equivalent.
  - Note: this fact has been formally demonstrated.

Limits of automated static analysis of code: examples

- We can easily find that a variable is used before being initialized in a case like this one:

```
int n;
if (n>0) ...
```

- On the contrary, we cannot be sure that the same problem occurs in a case like this one:

```
int n;
...  // some complex code here
if (n>0) ...
```

In general, we cannot predict if the execution of some complex code will result in initializing **n**.

Possible approaches to manage limits

- Often, a tool cannot establish with certainty that a problem is bound to occur.
- Therefore, tools can be
  - Rigorous: only errors that the tool is sure will occur are reported. The users must be aware that when the tool reports no errors, some errors may occur anyway.
  - Pessimist: all possible errors are reported. The users must be aware that not all reported errors are real errors, i.e., some may never occur.
- In both cases, users have to perform some work to “correct” the indications provided by the tool.

Tools for static code analysis

- Many tools are available
  - Several good open-source tools are available
- In general a tool support just one programming language (or a very small number of languages)

In this tutorial

- We consider only automated static analysis of code.
  - Hence, “static analysis” will refer to “automated static analysis of code”
- We shall use a few specific tools
- We shall analyse only Java code
- We shall look in some detail at some types of problems that tools can spot

Tools

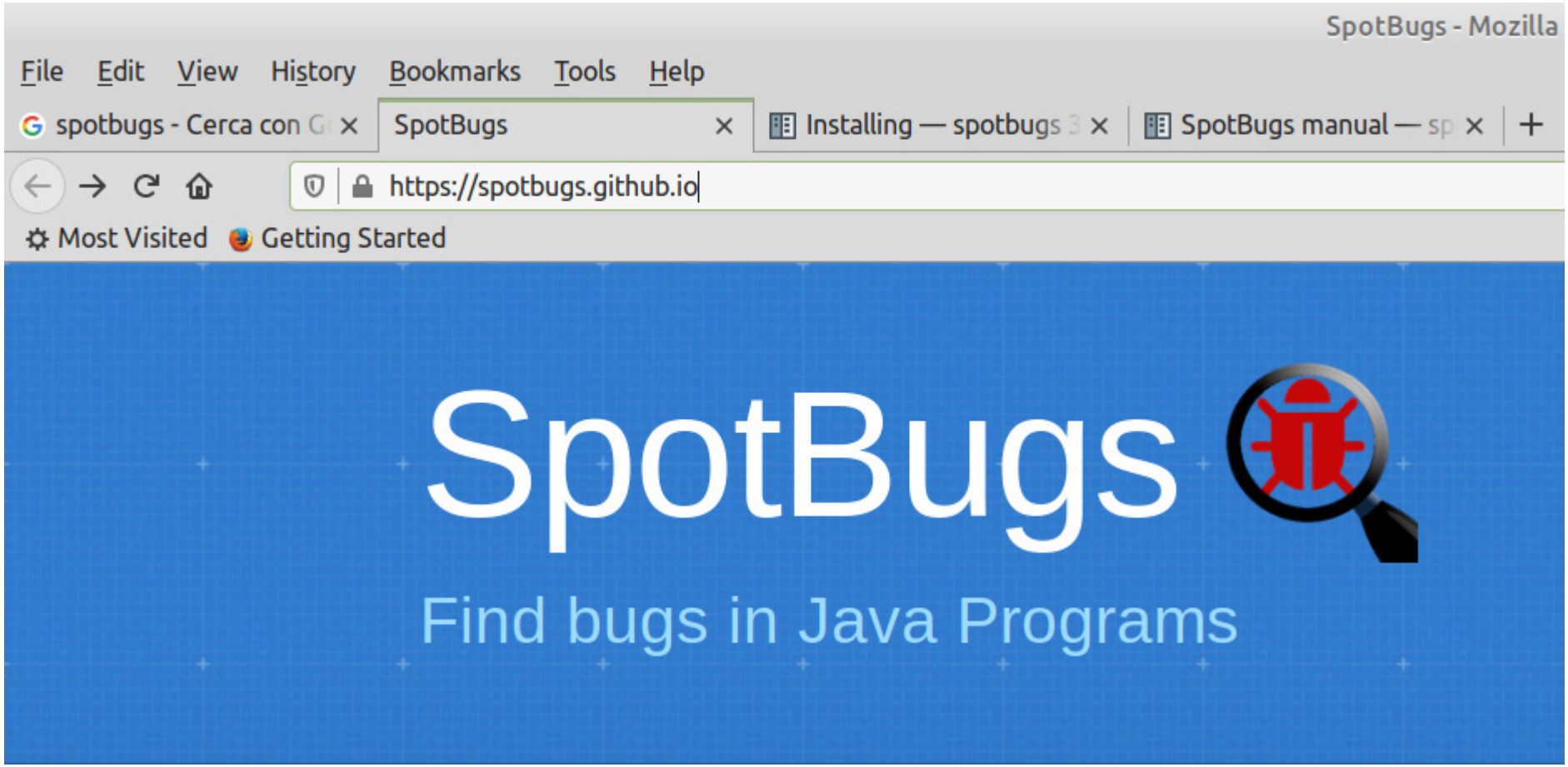
- A list is available at https://github.com/checkstyle/checkstyle/wiki/Java-static-code-analysis-tools
  - SpotBugs (https://spotbugs.github.io/)
    - Formerly known as Findbugs
  - PMD
  - Checkstyle
  - Lint4J
  - Classycle
  - Jdepend
  - SISSy
  - Google Codepro

In this tutorial

- We shall see SpotBugs at work
- I am using the Linux version.
  - The same functionality is available in Windows
- We shall use the stand-alone version
  - You can use SpotBugs from Maven and other environments

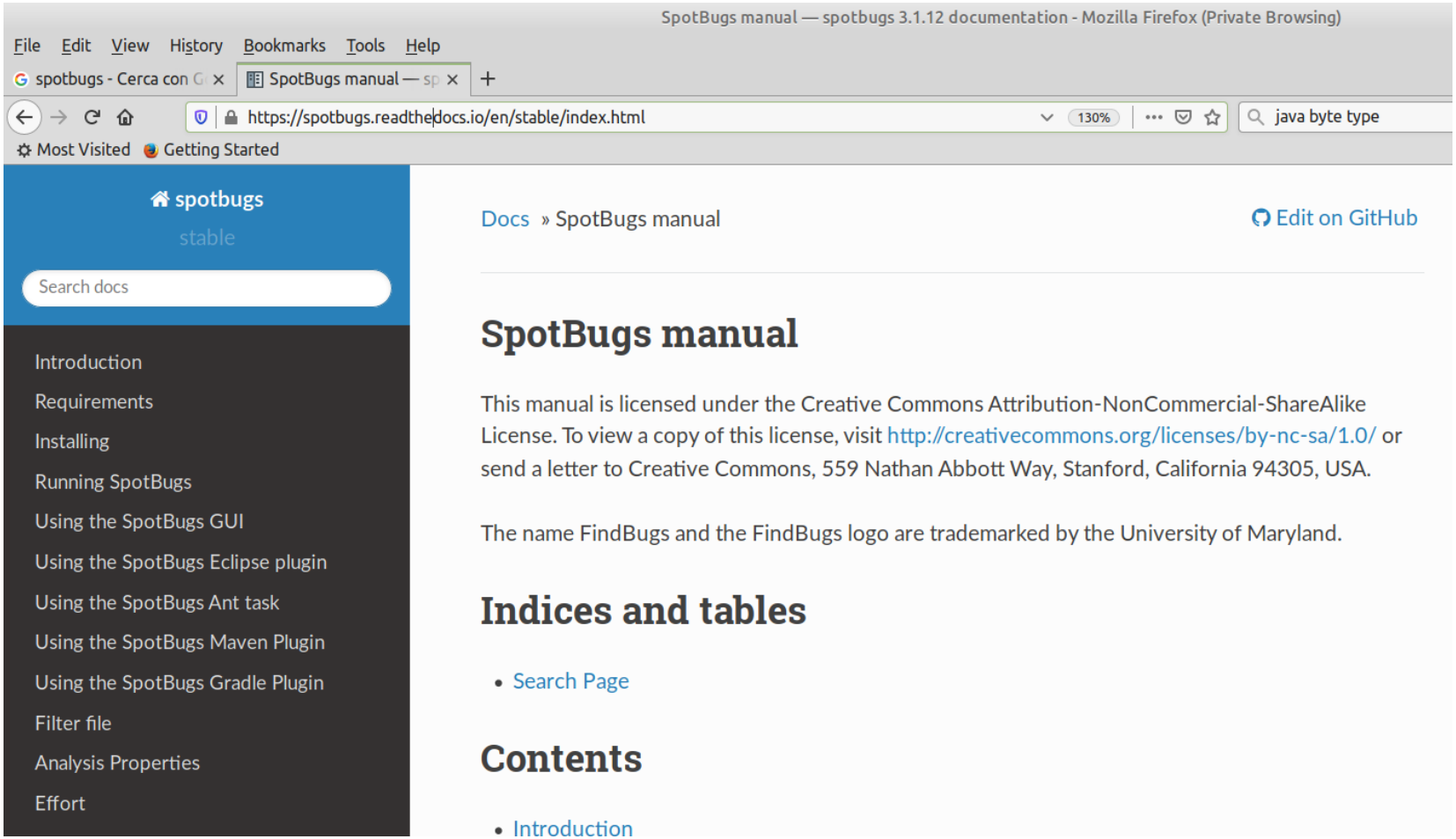
SpotBugs

- Spotbugs.github.io



SpotBugs documentation

- The documentation is available on line at https://spotbugs.readthedocs.io/en/stable/index.html



Download and installation

- See the documentation.
- You just have to download and unzip a file.

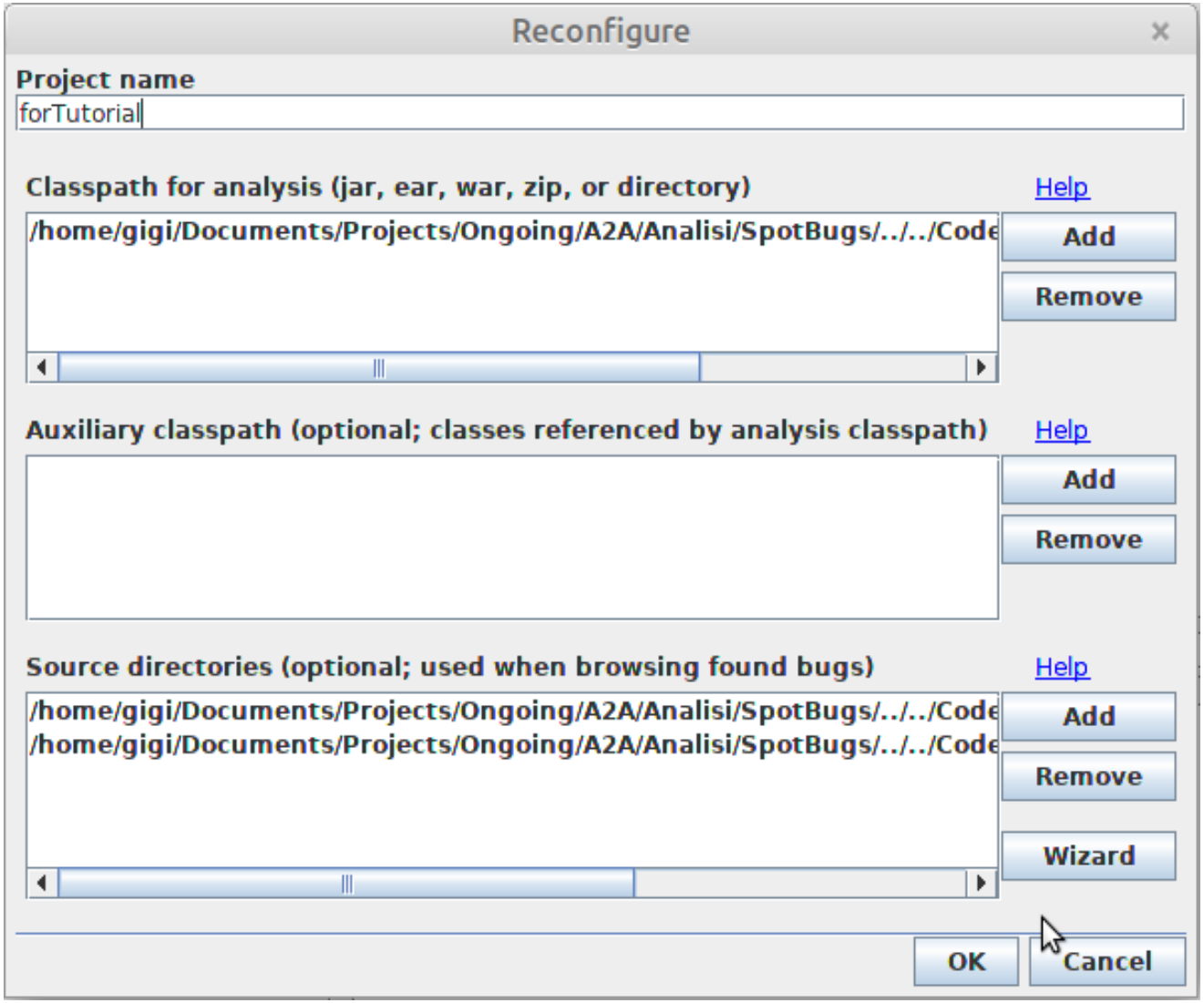
Before usage

- The tool analyses bytecode, hence you need to have the compiled code.
- The tool can visualize the code line where the problem was found
  - To this end, you need to provide the source code.

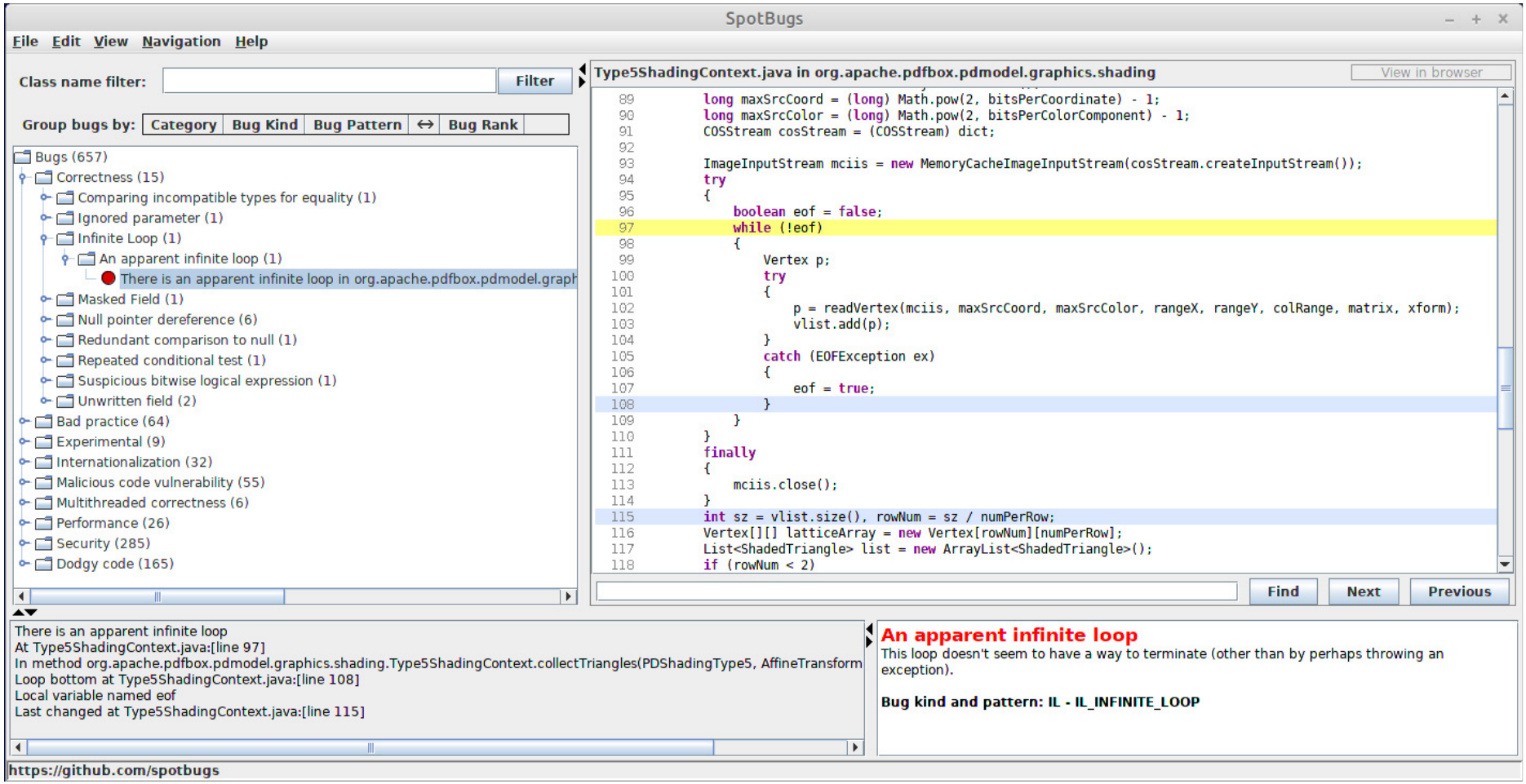
Usage (via demo)

- Project creation
- Analysis
- Browsing results
- Saving results

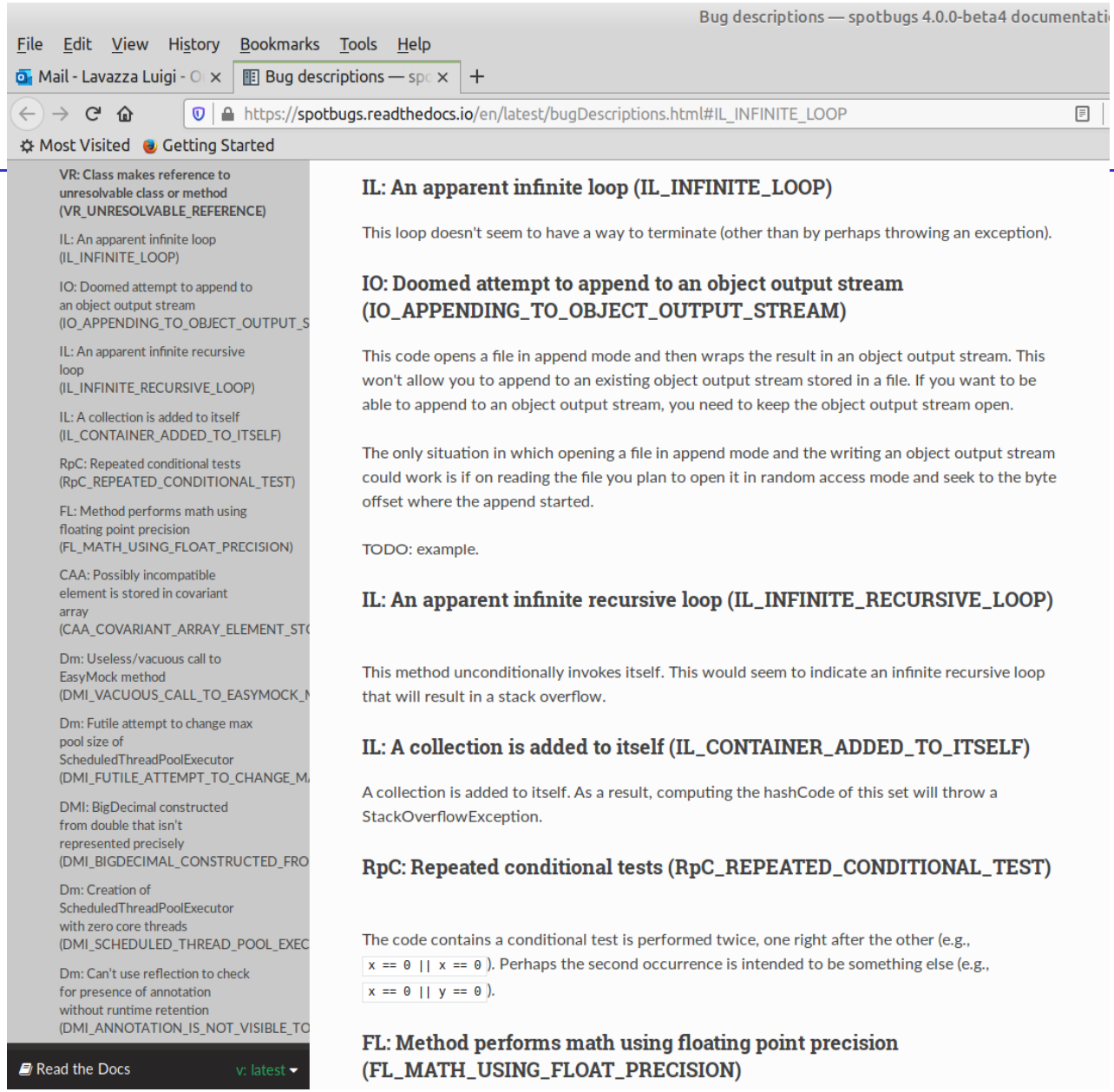
Spotbugs: analysis configuration



Analysis results



Bug explanation



Example of analysis

- Here we analyse Apache PDFBox® v. 2.0.14
- An open source library of Java tools for working with PDF documents.
  - https://pdfbox.apache.org/

Bug classes

- SpotBugs is able to detect several types of potential bugs.
- In our case, we get bugs in the following classes:
  - Correctness
  - Bad practice
  - Experimental
  - Internationalization
  - Malicious code vulnerability
  - Performance
  - Security
  - Dodgy code

BUG: Bitwise OR of signed byte value

- Loads a byte value (e.g., a value loaded from a byte array or returned by a method with return type byte) and performs a bitwise OR with that value. Byte values are sign extended to 32 bits before any bitwise operations are performed on the value. Thus, if b[0] contains the value 0xff, and x is initially 0, then the code ((x << 8) | b[0]) will sign extend 0xff to get 0xffffffff, and thus give the value 0xffffffff as the result.

BUG: Bitwise OR of signed byte value

- The following code for packing a byte array into an int is badly wrong:

```
int result = 0;
for(int i = 0; i < 4; i++) {
    result = ((result << 8) | b[i]);
}
```

- The following idiom will work instead:

```
int result = 0;
for(int i = 0; i < 4; i++) {
    result = ((result << 8) | (b[i] & 0xff));
}
```

In our case

```
public int getFamilyClass()
    {
        return bytes[0] << 8 | bytes[1];
    }
```

- Where **`bytes`** is an array of **`byte`**
- For us, it is hard to tell if this is an error. Probably, pdfbox developers could evaluate the situation easily.

BUG: Call to `equals()` comparing different types

- This method calls **`equals(Object)`** on two references of different class types and analysis suggests they will be to objects of different classes at runtime. Further, examination of the **`equals`** methods that would be invoked suggest that either this call will always return **`false`**, or else the **`equals`** method is not be symmetric (which is a property required by the contract for equals in class **`Object`**).

In our case

```
assertEquals("Null Value...", COSNumber.get(null));
```

where **`COSNumber.get`** is defined as follows:

```
public static COSNumber get( String number ) throws
IOException
```

This is clearly an error. The code compares a **`COSNumber`** with a **`String`**.

BUG: Non-virtual method call passes `null` for non-null parameter

- A possibly-null value is passed to a non-null method parameter. Either the parameter is annotated as a parameter that should always be non-null, or analysis has shown that it will always be dereferenced.

In our case

```
checkPerms(inputFileAsByteArray, "", null);

private void checkPerms(byte[] inputFileAsByteArray,
        String password,
        AccessPermission expectedPermissions) throws IOException
    {
        PDDocument doc = PDDocument.load(inputFileAsByteArray,
password);

        AccessPermission currentAccessPermission =
doc.getCurrentAccessPermission();

        // check permissions
        assertEquals(expectedPermissions.isOwnerPermission(),
currentAccessPermission.isOwnerPermission());
```

This is clearly an error. In the considered case, the code tries to execute **`isOwnerPermission`** of **`expectedPermissions`**, but **`expectedPermissions`** is **`null`**.

BUG: Repeated conditional tests

- The code contains a conditional test is performed twice, one right after the other (e.g., x == 0 || x == 0). Perhaps the second occurrence is intended to be something else (e.g., x == 0 || y == 0).

In our case

```
private void mergeFields(PDFCloneUtility cloner,
                PDField destField, PDField srcField)
    {
        if (destField instanceof PDNonTerminalField &&
srcField instanceof PDNonTerminalField)
        {
            LOG.info("Skipping non terminal field " +
srcField.getFullyQualifiedName());
            return;
        }

        if (destField.getFieldType() == "Tx" &&
destField.getFieldType() == "Tx")
```

Probably, the programmer intended to check the equality of **`destField`** and **`srcField`** types: a typical copy&paste error.

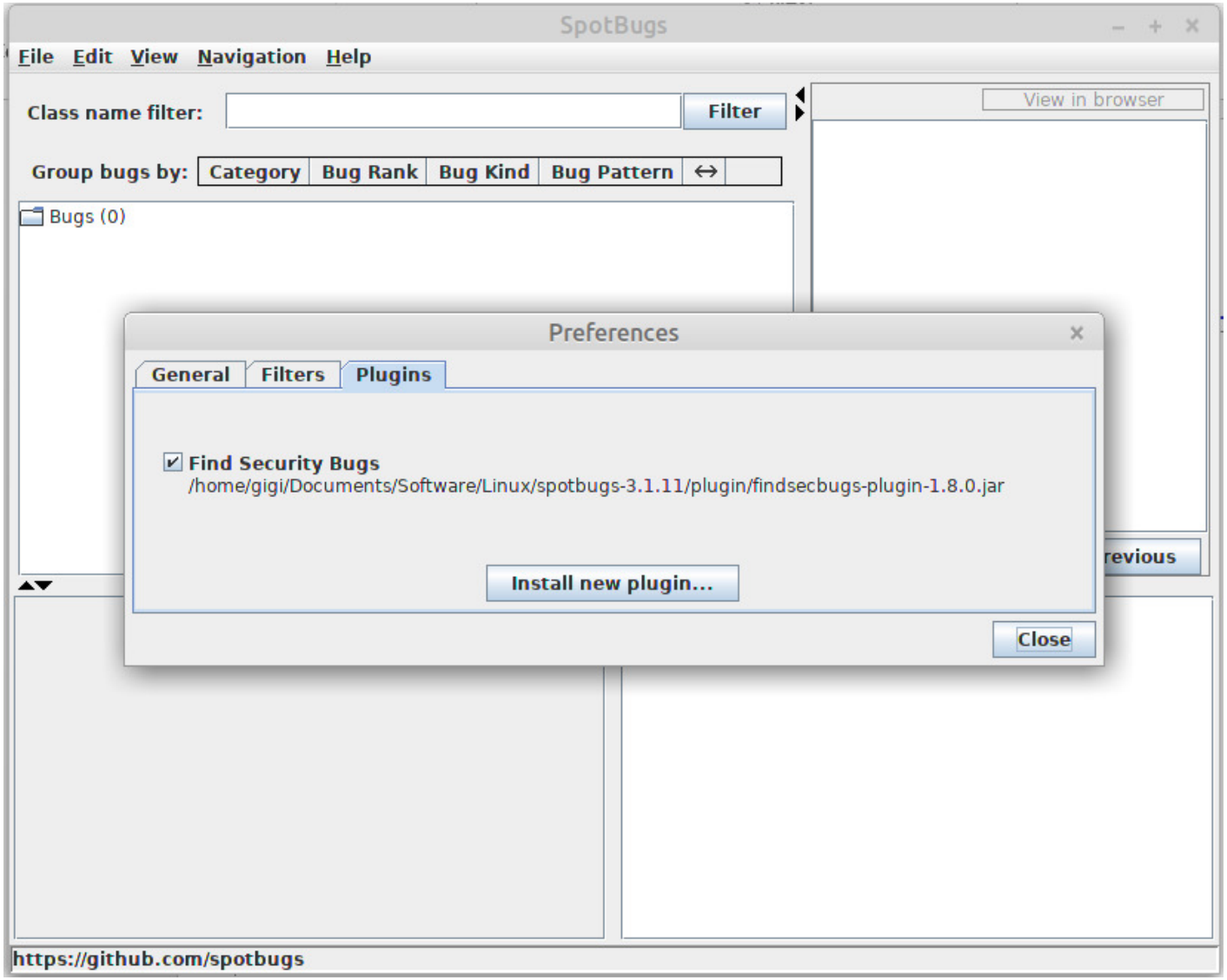
More bugs

- demo

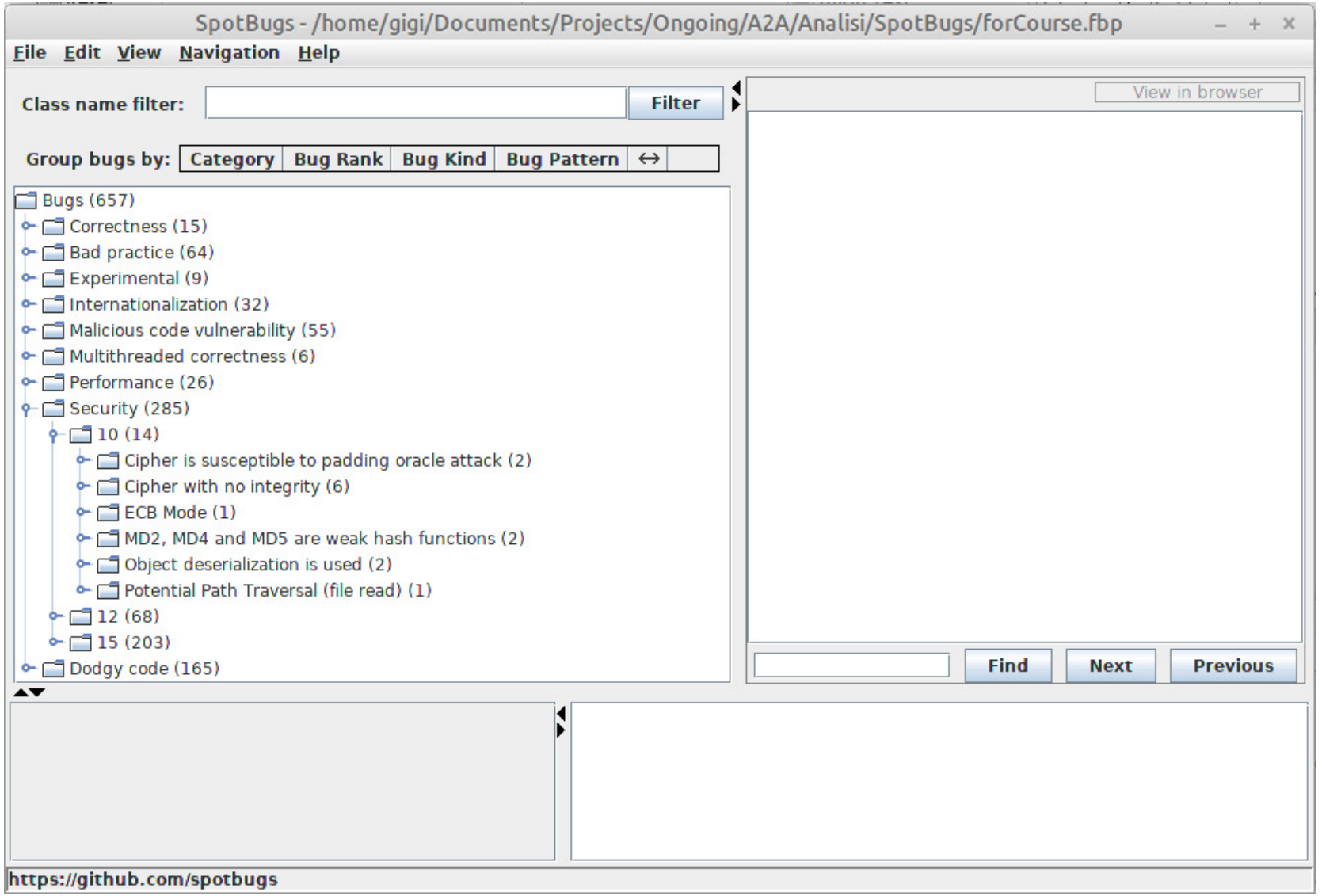
Security

- Several security problems (e.g., code vulnerabilities) can be identified via static analysis.
- SpotBugs can be used with a plug-in (SpotSecurityBugs) devoted specifically to find security bugs.

SpotSecurityBugs



SpotSecurityBugs



BUG: Cipher is susceptible to Padding Oracle

- This specific mode of CBC with PKCS5Padding is susceptible to padding oracle attacks. An adversary could potentially decrypt the message if the system exposed the difference between plaintext with invalid padding or valid padding. The distinction between valid and invalid padding is usually revealed through distinct error messages being returned for each condition.
- Code at risk:

```
Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

- Solution:

```
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

BUG: Cipher is susceptible to Padding Oracle

- References
  - Padding Oracles for the masses (by Matias Soler)
  - Wikipedia: Authenticated encryption
  - NIST: Authenticated Encryption Modes
  - CAPEC: Padding Oracle Crypto Attack
  - CWE-696: Incorrect Behavior Order

SpotSecurityBugs addresses well-known security bugs, which have been catalogued and described by authoritative organizations.

BUG: Cipher with no integrity

- The ciphertext produced is susceptible to alteration by an adversary. This mean that the cipher provides no way to detect that the data has been tampered with. If the ciphertext can be controlled by an attacker, it could be altered without detection.
- The solution is to used a cipher that includes a Hash based Message Authentication Code (HMAC) to sign the data. Combining a HMAC function to the existing cipher is prone to error [1]. Specifically, it is always recommended that you be able to verify the HMAC first, and only if the data is unmodified, do you then perform any cryptographic functions on the data.
- The following modes are vulnerable because they don't provide a HMAC: - CBC - OFB - CTR - ECB

BUG: Cipher with no integrity

- The following snippets code are some examples of vulnerable code.
- Code at risk:
  - AES in CBC mode

```
cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

  - Triple DES with ECB mode

```
cipher c = Cipher.getInstance("DESede/ECB/PKCS5Padding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

- Solution:

```
cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

- In the example solution above, the GCM mode introduces an HMAC into the resulting encrypted data, providing integrity of the result.

The reliability of static analysis

- In principle, static analysis tools may consider incorrect several situations that are actually correct.

- In practice, most of the bugs reported by SpotBugs are real bugs. And when they are not, quite often they reveal that code was not written in a very careful way.

Static code measures

- Code measures were proposed to identify weakness in code design and organization.
- Several code measures are correlated to important software qualities (like correctness and maintainability)
- Hence, looking at measures we can identify the pieces of code that could make maintainability harder, or that could make the software more error-prone, etc.
- In conclusion, we are interested in knowing the characteristics of our code, as described by a set of measures.

Static code measures

- Measures were defined for
  - Size
    - At different levels: system, class, method, ...
  - Complexity
    - At the function or method level
  - Coupling
    - At the class or subsystem level
  - Cohesion
    - At the class level
  - ...

Measures and quality

- A well modularized system is easier to maintain.
  - A well modularized system is characterized by modules (classes) that have high cohesion and are loosely coupled.
- A complex function (method) is more difficult to test and maintain
  - A complex function (method) is characterized by a large number of independent paths in the flow graph (McCabe's complexity).
- Large methods are more difficult to maintain
- ...

Tools for code measurement

- There are many, several open-source
- Here we use Sourcemeter
  - https://www.sourcemeter.com/
- Thorough documentation (covering also installation) is available at https://www.sourcemeter.com/resources/java/

SourceMeter

- A batch program: saves results in a set of files

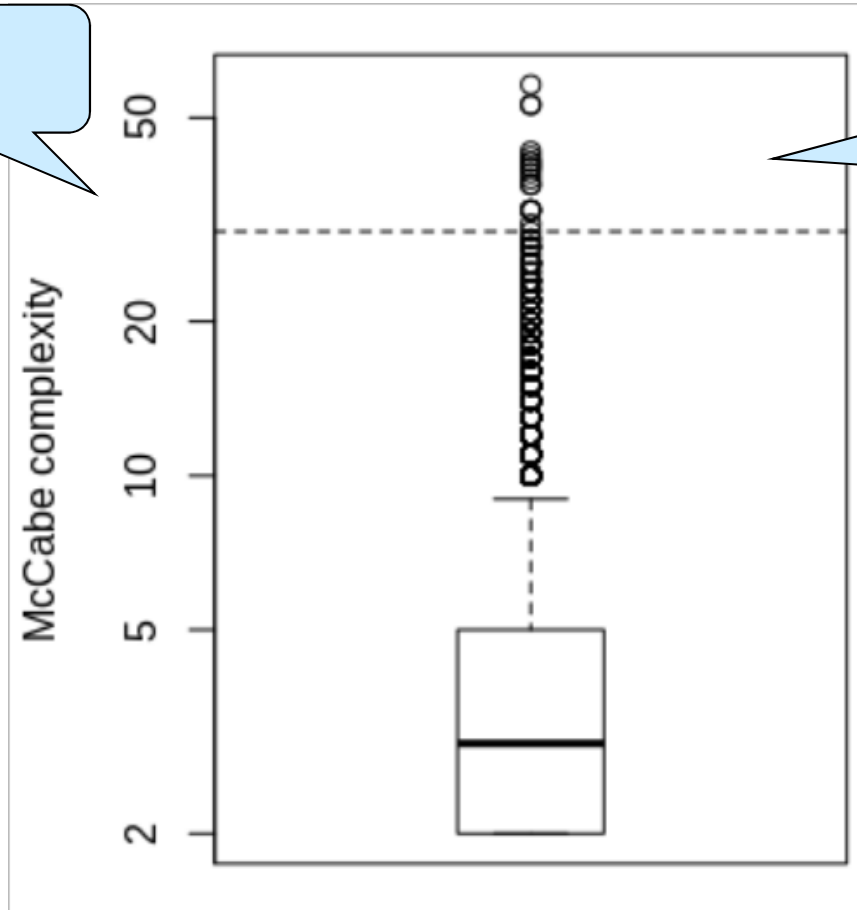
Name	Size	Type
sourcemeter	4 items	folder
pdfbox.graph	25,3 MB	unknown
pdfbox.xml	108,5 MB	XML document
pdfbox-Annotation.csv	2,6 kB	CSV document
pdfbox-Attribute.csv	2,1 MB	CSV document
pdfbox-Class.csv	973,0 kB	CSV document
pdfbox-CloneClass.csv	42,6 kB	CSV document
pdfbox-CloneInstance.csv	275,9 kB	CSV document
pdfbox-clones.txt	264,9 kB	plain text document
pdfbox-Component.csv	2,1 kB	CSV document
pdfbox-Enum.csv	27,4 kB	CSV document
pdfbox-File.csv	234,4 kB	CSV document
pdfbox-Folder.csv	28,8 kB	CSV document
pdfbox-Interface.csv	29,9 kB	CSV document
pdfbox-Method.csv	7,8 MB	CSV document
pdfbox-MetricHunter.txt	6,6 MB	plain text document
pdfbox-Package.csv	59,8 kB	CSV document
pdfbox-PMD.txt	1,4 MB	plain text document
pdfbox-RTEHunter.txt	0 bytes	plain text document
pdfbox-VulnerabilityHunter.txt	0 bytes	plain text document

How to use SourceMeter's results

- Most files are csv files: you can open them with a spreadsheet and perform any type of analysis the spreadsheet is able to support.
- Alternatively, you can read the file with a program you wrote and perform any processing you like.
- In what follows we shall see some processing performed via R programs.
  - https://www.r-project.org/

McCabe Complexity

Logarithmic scale!



There are several methods having CC > 30
These methods are likely to cause problems

- The distribution of McCabe complexity through PdfBox methods (excluding those having CC=1, e.g., setters and getters).

Excessively complex methods (CC≥30)

org.apache.xmpbox.DateConverter.toCalendar
org.apache.pdfbox.filter.Predictor.decodePredictorRow
org.apache.pdfbox.multipdf.PDFMergerUtility.appendDocument
org.apache.pdfbox.pdfparser.BaseParser.parseCOSString
org.apache.pdfbox.pdfparser.PDFStreamParser.parseNextToken
org.apache.pdfbox.pdfparser.PDFStreamParser.hasNoFollowingBinData
org.apache.pdfbox.tools.ExtractText.startExtraction
org.apache.pdfbox.tools.PDFToImage.main
org.apache.fontbox.afm.AFMParser.parseFontMetric
org.apache.fontbox.cff.Type1CharString.handleCommand
org.apache.fontbox.cff.Type2CharString.handleCommand
org.apache.fontbox.cmap.CMapParser.parseNextToken
org.apache.xmpbox.schema.AbstractXMPSchemaTest.testGetSetProperty
org.apache.pdfbox.pdmodel.fdf.FDFAnnotation.<init>
org.apache.pdfbox.pdmodel.fdf.FDFDictionary.<init>
org.apache.pdfbox.preflight.content.StubOperator.process
rchange.taggedpdf.PDLayoutAttributeObject.toString
org.apache.pdfbox.pdmodel.graphics.color.PDColorSpace.create
org.apache.pdfbox.pdmodel.graphics.image.CCITTFactory.extractFromTiff

Is high CC really dangerous?

- Let us look at the most complex method:
  `org.apache.pdfbox.multipdf.PDFMergerUtility.appendDocument`

- This method is quite long (380 LoC, 330 effective LoC)
- It has nesting level = 5

- In conclusion, this is mainly a long method.
- Splitting this method would make the code easier to test and maintain

False alarms

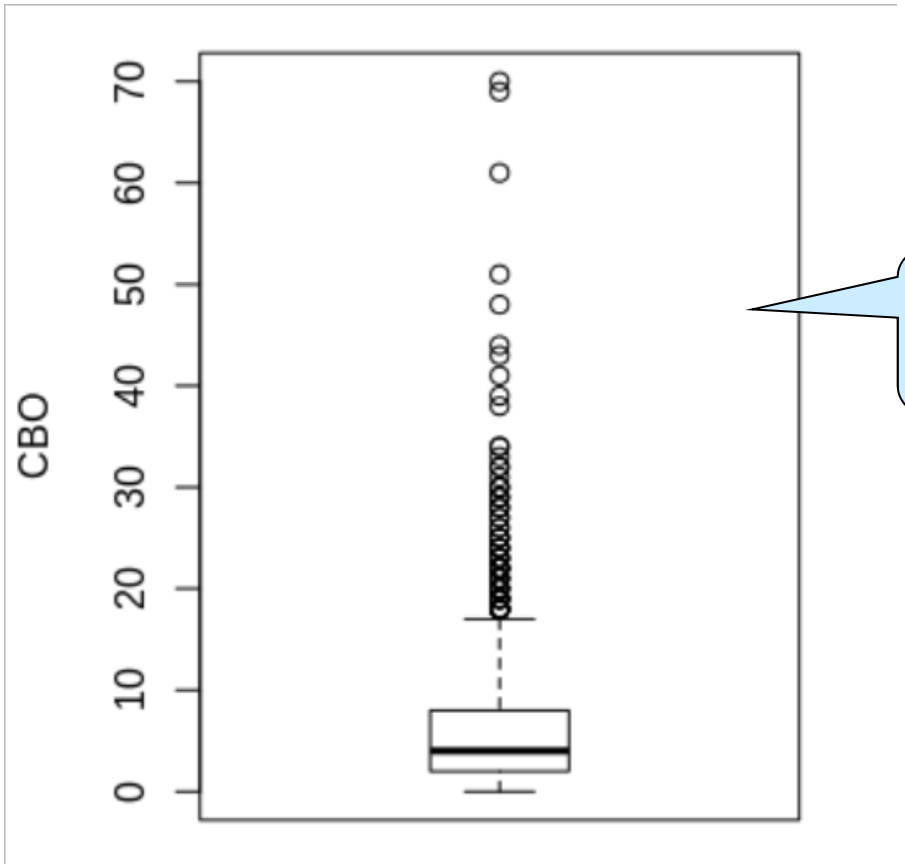
- The next more complex methods are
  - `org.apache.pdfbox.pdfparser.PDFStreamParser.parseNextToken`
  - `org.apache.fontbox.cmap.CMapParser.parseNextToken`
- These contain large switch statements. However, each case branch is relatively short (the largest ones fit in a screen).
- Hence, we do not need to worry about these methods.

Looking at classes

- Some measures should be particularly effective in spotting problems
  - CBO (coupling between objects)
  - RFC (response for class)
  - WMC (weighted methods per class)

Coupling

- The distribution of CBO through PdfBox classes.



Coupling above 30 is definitely suspicious.

Possibly critical classes

- The classes with highest CBO are
  - **`PDFMergerUtility`**
  - **`PDDocument`**
  - **`PageDrawer`**
- Noticeably, these classes also have
  - Large number of LoC
  - High WMC
  - High RFC

Possibly critical classes

- By examining the classes that have abnormal measures values, we find that
  - They are critical in that they are quite important.
    - They collect several basic functionalities
    - Hence, they are widely used
  - It is necessary to pay particular attention to these classes
    - They should be tested more accurately than other classes
    - Their design should be polished and maintained with care
    - ...

How to use measure in the development process

- In general, it is not possible to say that a measure being “too high” or “too low” implies that problems are likely.
- You have to “manually” inspect classes and methods whose measures appear abnormal.
- It is difficult to establish threshold such that a measure above a given thresholds should always be considered too high
  - Similarly for too low values
- You should continuously monitor the measures of classes and methods. When a measure of a given element changes substantially, you should investigate the reasons.

Code clones

- Quite often, programmers just copy and paste pieces of code that are needed in different parts of the system.
- This is not a good practice in general, since it generates the double maintenance problem.
  - If a piece of duplicated code is modified, usually it is necessary to modify its copies as well, but very often this is not done. The result is that you have inconsistency in code.
  - Even if all the copies are kept consistent, maintenance cost is duplicated.
- The solution consists in creating methods that can be called from different places in the code, or in introducing super-classes that let the same code become available in different sub-classes.

Defining code clones

- Identifying code clones can be difficult, because in general we do not look simply for pieces of code that are identical to each other.
- We also want to spot pieces of code that are “very similar”.

- Every tool supports its own interpretation of “very similar”.
- You have to check the definition supported by the tool you are using.
  - Sometimes parameters are available to customize the concept of code clone.

SourceMeter and code clones

- SourceMeter is able to detect code clones.

- Example:

```
1759~CloneClass [Number of Clone Instances: 2, Lines of Code: 18]
org\apache\pdfbox\rendering\PDFRenderer.java(494):
1760~CloneInstance [Ln:494, Col:9 - Ln:513, Col:10]
org\apache\pdfbox\rendering\PageDrawer.java(1818):
1761~CloneInstance [Ln:1818, Col:9 - Ln:1835, Col:10]
```

An example of similar blocks of code

- SourceMeter finds several similar blocks of code throughout the system.
- Specifically, it finds similar blocks in
  - Method **`hasBlendMode`** in class **`PageDrawer`**
  - Method **`hasBlendMode`** in class **`PDFRenderer`**

Method hasBlendMode in class PageDrawer

```
private boolean hasBlendMode(PDTransparencyGroup group, Set<COSBase> groupsDone) {
        if (groupsDone.contains(group.getCOSObject())) {
            // The group was already processed. Avoid endless recursion.
            return false;
        }
        groupsDone.add(group.getCOSObject());
        PDResources resources = group.getResources();
        if (resources == null)
            return false;
        }
        for (COSName name : resources.getExtGStateNames()) {
            PDExtendedGraphicsState extGState = resources.getExtGState(name);
            if (extGState == null) {
                continue;
            }
            BlendMode blendMode = extGState.getBlendMode();
            if (blendMode != BlendMode.NORMAL) {
                return true;
            }
        }
        // Recursively process nested transparency groups
        for (COSName name : resources.getXObjectNames()) {
          // omissis
        }
        return false;
    }
```

Method hasBlendMode in class PDFRenderer

```
private boolean hasBlendMode(PDPage page){
        // check the current resources for blend modes
        PDResources resources = page.getResources();
        if (resources == null){
            return false;
        }
        for (COSName name : resources.getExtGStateNames()){
            PDExtendedGraphicsState extGState = resources.getExtGState(name);
            if (extGState == null){
                // can happen if key exists but no value
                // see PDFBOX-3950-23EGDHXSBBYQLKYOKGZUOVYVNE675PRD.pdf
                continue;
            }
            BlendMode blendMode = extGState.getBlendMode();
            if (blendMode != BlendMode.NORMAL){
                return true;
            }
        }
        return false;
    }
```

Managing code clones

- In the case of methods **`hasBlendMode`**, developers should consider introducing a service method that can be used by both **`PageDrawer`** **`PDFRenderer`** classes, in substitution of the duplicated code.
- Developer could also consider placing the two classes in a generalization hierarchy, so that the needed code can be inherited from a common class.

Same class code clones

- The previous example concerned a piece of code duplicated in different classes.
- It is also possible to find clones in the same class.
- Example:

```
1628~CloneClass [Number of Clone Instances: 15, Lines of Code: 12]
org\apache\xmpbox\type\ResourceRefType.java(92): 1629~CloneInstance [Ln:92, Col:5 - Ln:103, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(110): 1630~CloneInstance [Ln:110, Col:5 - Ln:121, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(128): 1631~CloneInstance [Ln:128, Col:5 - Ln:139, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(146): 1632~CloneInstance [Ln:146, Col:5 - Ln:157, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(164): 1633~CloneInstance [Ln:164, Col:5 - Ln:175, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(182): 1634~CloneInstance [Ln:182, Col:5 - Ln:193, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(200): 1635~CloneInstance [Ln:200, Col:5 - Ln:211, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(218): 1636~CloneInstance [Ln:218, Col:5 - Ln:229, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(236): 1637~CloneInstance [Ln:236, Col:5 - Ln:247, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(254): 1638~CloneInstance [Ln:254, Col:5 - Ln:265, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(272): 1639~CloneInstance [Ln:272, Col:5 - Ln:283, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(290): 1640~CloneInstance [Ln:290, Col:5 - Ln:301, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(308): 1641~CloneInstance [Ln:308, Col:5 - Ln:319, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(326): 1642~CloneInstance [Ln:326, Col:5 - Ln:337, Col:6]
org\apache\xmpbox\type\ResourceRefType.java(344): 1643~CloneInstance [Ln:344, Col:5 - Ln:355, Col:6]
```

- All these clones are in class **`ResourceRefType`**

Clones in class ResourceRefType

```
public String getDocumentID() {
        TextType absProp = (TextType) getFirstEquivalentProperty(DOCUMENT_ID, URIType.class);
        if (absProp != null) {
            return absProp.getStringValue();
        }
        else {
            return null;
        }
    }

    public String getFilePath() {
        TextType absProp = (TextType) getFirstEquivalentProperty(FILE_PATH, URIType.class);
        if (absProp != null) {
            return absProp.getStringValue();
        }
        else {
            return null;
        }
    }

    . . .
    . . .
```

Clones in class `ResourceRefType`

- In this case the clones are due to a bad programming practice.
- Duplicating the code can easily be avoided by defining a single method

```
public String getProperty(String PropertyName) {
   TextType absProp = (TextType)
           getFirstEquivalentProperty(PropertyName, URIType.class);
   if (absProp != null) {
       return absProp.getStringValue();
   }
   else {
       return null;
   }
}
```

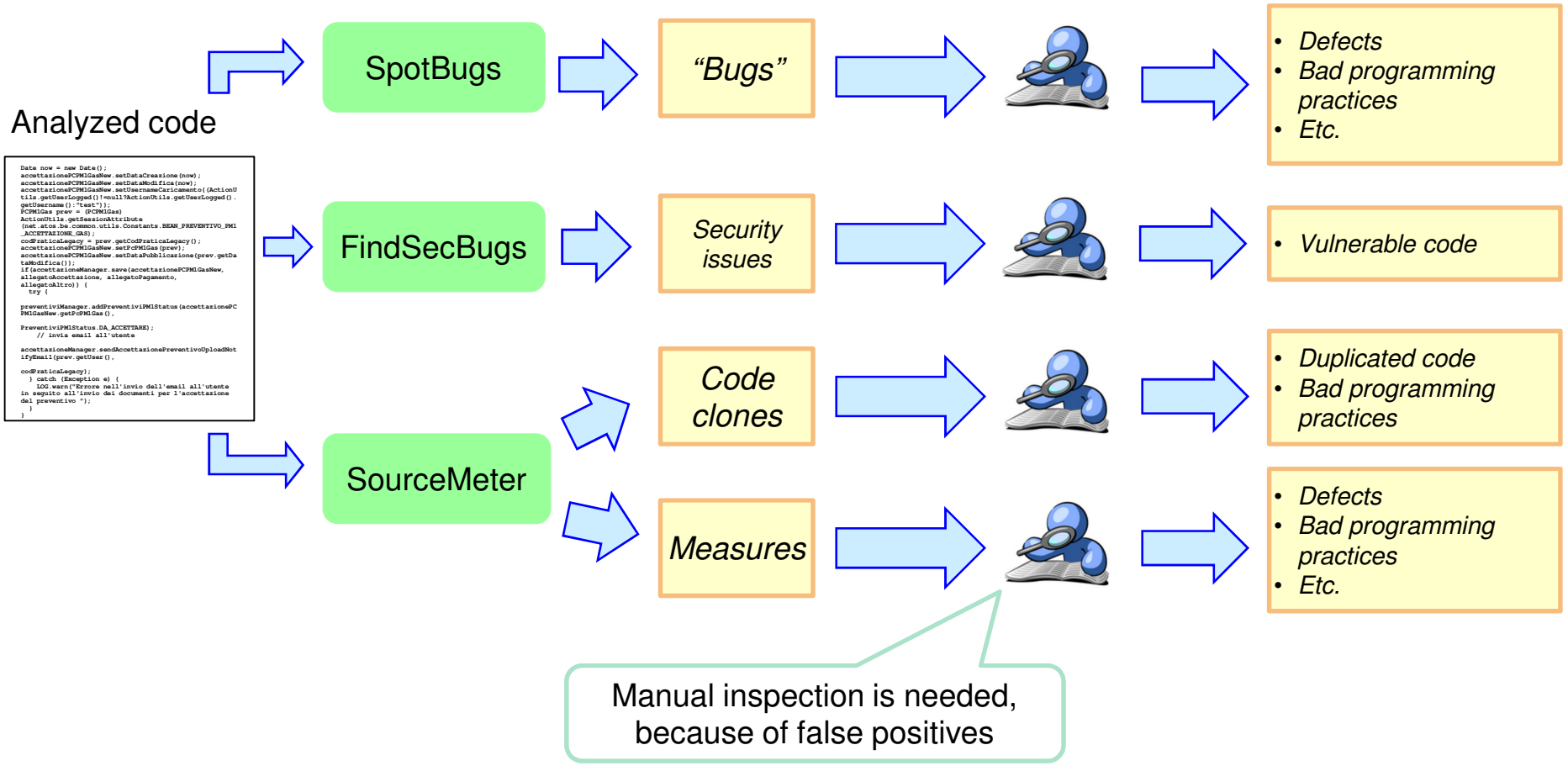
- Give this method, calling `getFilePath()` is equivalent to calling `getProperty(FILE_PATH)`, calling `getDocumentID()` is equivalent to calling `getProperty(DOCUMENT_ID)`, etc.

Clones in class `ResourceRefType`

- If–for some reason–you need to keep methods **`getFilePath()`**, **`getDocumentID()`**, etc. because you need to expose them in the class interface, you can introduce a private method that performs the core of the work, and each exposed method calls the private method.

Concluding remarks

The complete picture



Static code analysis in the development process

- We saw that a tool like SpotBugs can provide useful indications concerning the quality of code.
- When and how should we use such indications?
- Since the cost of removing bugs increases in time (the later you remove a bug, the more expensive) we want to remove bugs as soon as possible.
- Hence, programmers should run SpotBugs before they release a new version of their software.
- As an additional measure, SpotBugs should be used before starting testing.
  - Suppose SpotBugs finds several bugs in a new version of the application to be tested: it does not make sense to perform testing and find a lot of bugs we already knew were there.

Conclusions

- Static analysis is easy and cheap
  - Although in general you have to check results manually
- It can be nicely integrated in the development process
- Good open-source tools are available
- It can save a lot of time and money by spotting bugs as soon as the code is written.

- I do not see any good reason for not using static analysis on a regular basis.

THANK YOU FOR YOUR ATTENTION!