Lecture 30 — March 20, 2017

Patrick Lam version 1

Additional note about "fail fast" from last week: often, software reads configuration in the initialization phase and then acts on it later. Example: a monitoring system calls "dial" executable to report an issue. It's better if the monitoring system reports that its path for "dial" doesn't work upon first load, rather than after there is an issue. Because if "dial" isn't there, you have the original issue, but you're probably not looking at the system, so the system can't tell you that there's that issue. Plus "dial" not being there is another issue.

Static code analysis: PMD

We'll dive into static code analysis by talking about one particular code analysis tool, PMD[1]. Last week, we said that it was better to use tools to flag style issues. PMD is one way to do so.

PMD out of the box: built-in rulesets

The easiest way to use PMD is in an IDE with its built-in rulesets. It has rulesets for languages from C++ to Scala, including Java. For Java there are a number of rulesets, which group related rules. Let's look at a few examples of rules.

- SimplifyConditional: [design ruleset] detect redundant null checks

```
1 class Foo {
2   void bar(Object x) {
3     if (x != null && x instanceof Bar) {
4       // just drop the "x != null" check
5     }
6   }
7 }
```

  Note that this code is not wrong. It's just redundant.
- UseCollectionIsEmpty: [design ruleset] better to use `c.isEmpty()` rather than `c.size() == 0`

```
1 class Foo {
2     void good() {
3         List foo = getList();
4         if (foo.isEmpty()) { /* blah */ }
5     }
6
7     void bad() {
8         List foo = getList();
```

[1] `pmd.github.io`

```
9           if (foo.size() == 0) { /* blah */ }
10      }
11  }
```

Again, it's not wrong to call `size()` and see if the result is 0. It's just more idiomatic, and sometimes more efficient, to check `isEmpty()`.

- MisplacedNullCheck: [basic ruleset] don't check nullness after relying on non-nullness

```
1  public class Foo {
2      void bar() { if (a.equals(baz) || a == null) {}     }
3  }
```

The check `a == null` is never going to succeed, because `a.equals()` would throw a `NullPointerException` instead. So if `a` can ever be null, there is a fault.

- UseNotifyAllInsteadOfNotify: [design ruleset] most of the time, `notifyAll()` is the right call to use, not `notify()`. Unless you know what you're doing, using `notify()` is going to result in a bunch of stuck threads, which is a bug.

Find more about the above rules at `https://pmd.github.io/pmd-5.5.4/pmd-java/rules/java/design.html`. The pages on the PMD site are also useful for your assignment as sample code.

I've included examples from the design and basic rulesets. There's a ruleset specifically for JUnit, rule sets detecting empty or otherwise useless code, naming conventions, and much more.

Linking back to last week's material: PMD and tools like it can tell you about things that may be wrong, or that are certainly wrong. However, they cannot tell you about how important that wrongness is. We still need (experienced!) human judgment to know that.

Writing your own PMD rules

Assignment 3 Question 1 asks you to write your own PMD rule. So we'll talk about how to write PMD rules. The intellectual core of a PMD rule is a query on the Abstract Syntax Tree (AST). You can use either Java or XPath to describe this query. XPath is cleaner, in that it's a declarative query language.

Here are some links about how to make rule sets and the boilerplate you need for rules:

- `https://pmd.github.io/latest/customizing/howtomakearuleset.html`
- `https://pmd.github.io/latest/customizing/howtowritearule.html`

We'll be focussing on what goes into the rule itself, as per `https://pmd.github.io/latest/customizing/xpathruletutorial.html`. The tutorial skips a lot of detail about how to actually use XPath. So let's start with that.

XPath. Let's start from the fundamentals. You write selectors to find nodes. We'll look at a simple XML document. XPath also applies to web programming (in particular the Document Object Model) and also to the Java code we'll be analyzing. Source: `https://www.w3schools.com/xml/xpath_syntax.asp`; specification: `https://www.w3.org/TR/xpath/`.

Here is an XML file:

```
<?xml version="1.0" encoding="UTF-8"?>

<bookstore>
 <book>
  <title lang="fr">Harry Potter</title>
  <price>29.99</price>
 </book>
 <book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
 </book>
</bookstore>
```

You can observe the tree structure of the file. And you can play along with either `https://www.w3schools.com/xml/tryit.asp?filename=try_xpath_select_cdnodes` (which is hardcoded to XML similar to the above) or else `http://www.freeformatter.com/xpath-tester.html`.

Consider XPath expression `//price`. The result is the set of price nodes with data `29.99`, `39.95`. So, expression `//price` selects nodes with name `price`; the `//` means any descendants (including self) of the context node (= root node, here)—we asked for all descendants of the root named `price`. And, `count(//price)` counts the number of `price` elements in the tree.

We can also specify an exact path through the tree, say with `/bookstore/book[1]/title`. This starts at the root, visits the `bookstore` element, then its first `book` child, then returns the title. If we omitted `[1]`, then we'd get all of the titles.

We can also select all elements that satisfy some condition, e.g. `/bookstore/book[price>35]/title` selects the titles of books with price greater than 35.

Note the `lang` attribute. We can select elements with a certain value for `lang`: `//title[@lang="fr"]`.

In general, square brackets can contain predicates. We've seen pretty simple ones, but you can put arbitrary tree queries, e.g. `//price[../title[text()="Learning XML"]]`.

You can also combine predicates with `and`, `or`, etc. e.g. `//title[../price < 35 or @lang="en"]`. `*` works as you might expect.

The double-slash `//` includes descendants and self. If you want descendants excluding self, write e.g. `descendant::book` as part of your expression. For the above example, there's no difference, but you can see it here:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
 <book>
  <book><title lang="fr">Harry Potter</title><price>29.99</price></book>
 </book>
 <book><title lang="en">Learning XML</title><price>39.95</price></book>
</bookstore>
```

Try `//book[descendent::book]` versus `//book[book]`.