Software Testing, Quality Assurance and Maintenance Winter 2017

Lecture 31 — March 22, 2017

Patrick Lam *version 1*

Using XPath to write PMD Rules

Last time, we saw how some notions on how to use XPath in general. Today, we'll talk about XPath and PMD.

Here's a rule ensuring that while statements must not contain directly contain other statements:

```
//WhileStatement[not(Statement/Block)]
```

This rule matches `WhileStatement`s whose `Statement` child does not contain a `Block`.

See `https://pmd.github.io/latest/customizing/xpathruletutorial.html` for more examples. Let's consider this example, detecting definitions of `Logger` variables.

```
1    public class a {
2        Logger log1 = null;
3        Logger log2 = null;
4        int b;
5
6        void myMethod() {
7            Logger log = null;
8            int a;
9        }
10       class c {
11           Logger a;
12           Logger b;
13       }
14   }
```

First, we can detect `Logger` variables. Note the use of `@Image`. We are matching `VariableDeclarator`s whose type is `Logger`:

```
//VariableDeclarator[../Type/ReferenceType/ClassOrInterfaceType[@Image='Logger']]
```

But we really want class definitions which contain more than one variable of type `Logger`.

```
//ClassOrInterfaceDeclaration
    [count(//VariableDeclarator
      [../Type/ReferenceType/ClassOrInterfaceType[@Image='Logger']])>1]
```

You can use the PMD Designer to debug your rules, as I'll demo in class. The PMD Designer allows you to browse the AST and try out your queries as you build them.

Hints for Assignment 3. I think I've talked about everything you need to use, except for the `starts-with` function that you can use for predicates. Everything else you can find from the PMD Designer. Recall that you're matching test methods that do *not* have a call to `mockCommandSender.getLastMessage` in an assert.

Using Linters

We will also talk about linters in this lecture, based on Jamie Wong's blog post `jamie-wong.com/2015/02/02/linters-as-invariants/`.

First there was C. In statically-typed languages, like C,

```c
#include <stdio.h>

int main() {
  printf("%d\n", num);
  return 0;
}
```

the compiler saves you from yourself. The guaranteed invariant:

"if code compiles, all symbols resolve."

Less-nice languages. OK, so you try to run that in JavaScript and it crashes right away. Invariant?

"if code runs, all symbols resolve?"

But what about this:

```javascript
function main(x) {
  if (x) {
    console.log("Yay");
  } else {
    console.log(num);
  }
}

main(true);
```

Nope! The above invariant doesn't work.

OK, what about this invariant:

"if code runs without crashing, all symbols referenced in the code path executed resolve?"

Nope!

```javascript
function main() {
  try {
    console.log(num);
  } catch (err) {
    console.log("nothing to see here");
  }
}

main();
```

So, when you're working in JavaScript and maintaining old code, you always have to deduce:

- is this variable defined?
- is this variable always defined?
- do I need to load a script to define that variable?

We have computers. They're powerful. Why is this the developer's problem?!

```
1  //jshint undef:true, devel:true
2
3  function main(x) {
4    "use strict";
5    if (x) {
6      console.log("Yay");
7    } else {
8      console.log(num);
9    }
10 }
11
12 main(true);
```

Now:

```
$ jshint lintee.js
lintee.js: line 8, col 17, 'num' is not defined.

1 error
```

Invariant:

"If code passes JSHint, all top-level symbols resolve."

Strengthening the Invariant. Can we do better? How about adding a pre-commit hook?

"If code is checked-in and commit hook ran,
all top-level symbols resolve."

Of course, sometimes the commit hook didn't run. Better yet:

- Block deploys on test failures.

Better invariant.

"If code is deployed,
all top-level symbols resolve."

Even better yet. It is hard to tell whether code is deployed or not. Use git feature branches, merge when deployed.

"If code is in master,
all top-level symbols resolve."