CS1632, Lecture 17: Static analysis, Part 1

Wonsun Ahn

Dynamic vs Static Testing

- Dynamic test – Code is executed by the test
  - Almost everything that we have done so far!

- Static test – Code is not executed by the test
  - Defect is found through analysis of code

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- Code metrics
- Linters
- Bug finders
- Formal verification

Why Static Test?

- Often easier than dynamic testing
    - What's easier than compiling code?
- Can pinpoint a defect better than a dynamic test can
    - A dynamic test just tells you there is a defect with a certain input
    - A static test analyzes the code and tells you exactly which line of code to fix
- Can often find defects that dynamic testing would miss
    - Dynamic testing is limited by its test cases – may miss certain behavior
    - A static test can (in theory) analyze the entire code thoroughly

Why not (only) Static Test?

- Often does not find all defects
  - E.g. just because a program compiles, doesn’t mean it is bug free!
  - E.g. just because you did a code review, doesn’t mean it is bug free!
  - With formal verification, you can catch *all* defects for certain programs – but more on that later
- Often reports false positives
  - False positive – as in the test reports a defect but it turns out there is none
  - E.g. you thought you found a bug through a code review, but it wasn’t a bug
  - Even automated tools like linters and bug finders are prone to false positives

Kinds of Static Tests

- Code review / walk-through – Eyeballing your code, next!
- Compiling
- Code coverage
- Code metrics
- Linters
- Bug finders
- Formal verification

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- Code metrics
- Linters
- Bug finders
- Formal verification

Compiler

- First job of compiler is to translate source code to machine code
- Second job is to perform static checks on source code
  - Errors – code does not adhere to language rules
    - Syntax errors: Compiler cannot parse code – structural problems
    - Type errors: Storing values into variables not meant for that data type
    - Uncaught exceptions
  - Warnings – code adheres to language rules but looks suspicious
    - Uninitialized variable – why use an unknown value?
    - Unused variable – did you forget to use this variable?
    - Dead code (unreachable code) – then why did you write it?
    - Implicit type conversion – may change the value too, implicitly

Compiler – Use it to the fullest!

- Warnings are their weight in gold
  - Programmers fix errors but tend to ignore warnings because it compiles
  - The compiler is trying to tell you something valuable, why ignore it?
- Let your compiler do static checking to the fullest
  - In gcc, "-Wall" command line option turns on all warnings
  - In most scripting languages, there is "use strict;" and/or "use warnings;"
    - JavaScript, Python, Perl, …
    - Put at top of source code enables more strict static checking
- New languages designed for stricter static checking
  - TypeScript: JavaScript with optional data type specifications
  - Rust: C with additional checks for memory safety

Choice of Language

- Even before writing a single line of code a lot is decided by …
  … your choice of programming language
  - How many defects your program is likely to have per LOC
    - With automatic memory management in language, less memory bugs
  - How many security vulnerabilities it is likely to have per LOC
    - If your language is sandboxed (e.g. Java), will have less security problems
  - What kind of performance problems the code will have
    - In JavaScript / Python, certain patterns trigger 10x slowdowns
    - In garbage collected languages, GC is part of the performance problem
- With good language, compiler can static test some of above problems

Choice of Language

- Language also decides how effective other static tests will be
  - Doing code review on assembly language is not fun
  - Automated tools often rely on semantic knowledge exposed by language
    - The more tool learns about program, the more it can check!
    - Data type in language tells tool a lot about a variable:
      E.g. if variable is of Reference type, tool checks for a different set of bugs, compared to when variable is of String type

- If program is a house, choice of language is the location
  - Language! Language! Language!

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- Code metrics
- Linters
- Bug finders
- Formal verification

Code Coverage

- How much of the codebase is covered by a particular test suite.
- You need to execute a test suite so isn’t this dynamic testing?
  - Yes, but a fair bit of static analysis is required to measure code coverage
  - Involves analyzing code and instrumenting with counters before running (e.g. How many times method called?)

- Code coverage can mean different things though!

Code Coverage

Consider following code and (pseudocode) unit tests

```
class Duck {
   public String quack(int x) {
      if (x > 0) {
         return "Quack!";
      } else {
         return “Negative Quack!”;
      }
   }
   public String quock() {
    return "Quock!";
   }
}


assertEquals(quack(1), “Quack!”);
assertEquals(quack(-4), “Negative Quack!”);
```

Method Coverage

- What percentage of all methods have been called?
- In previous example, 50%

Code Coverage

Consider following code and (pseudocode) unit tests

```java
public static int noogie(int x) {
   if (x < 10) {
      return 1;
   } else {
      if ((int) Math.sqrt(x) % 2 == 0) {
         return (x / 0);
      } else {
         return 3;
      }
   }
}

assertEquals(noogie(5), 1);
assertEquals(noogie(81), 3);
assertEquals(noogie(9), 3);
```

Statement Coverage

- Percentage of statements that have been tested
- That’s 100% method coverage, but we are missing some statements!
- This is usually what’s referred to as “code coverage” (although technically it’s a *kind* of code coverage)

Other Kinds of Code Coverage

- Branch coverage: every branch direction (e.g. if statement) taken?
- Condition coverage: every Boolean expression tested true/false?
- Path Coverage: every possible path through method taken?
- Parameter value coverage: every common parameter value covered?
- Entry/Exit Coverage: every method call / return covered?
- State coverage:
  - Every state covered if program expressed as finite state machine?
  - Arguably the best definition of coverage but hard to measure:
    1) Only simple programs have a finite number of states
    2) First the program has to be expressed as an FSM

What does Code Coverage tell you?

- Where more tests would be useful and where tests are missing
- Other than state coverage, 100% coverage does not mean defect free
- Consider the following…

```java
public int chirp(int x, int y) {
   double z = Math.sqrt(x);
   return (int) z / y;
}

// 100% (statement) coverage! WOO-HOO!

assertEquals(chirp(1, 1), 1);
```

Note

- Low coverage is bad, but high coverage does not always mean good.
- Even 100% of (statement) coverage cannot catch 100% of bugs!

Things Code Coverage Can’t Catch

- Different input values
- Combinatorial issues (different combinations of input values)
- Race conditions, or any other Heisenbug (nondeterministic) bugs
  - Exact same input values with different results from run to run
- More!

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- Code metrics
- Linters
- Bug finders
- Formal verification

Code Metrics

- Allows you to check :
  - Cyclomatic complexity!
  - Class fan-out!
  - Number of lines per class!
  - Number of interfaces!
  - Depth of inheritance tree!
  - NORM (Number of overridden methods)
  - Weighted methods per class!
- Some meta-data about code that measures complexity
  - Premise: more complexity leads to more defects

Code Metrics

- Honestly, I have never found these very useful.
- Some people/companies swear by them, though.
- You can set “triggers” for these in checkstyle.

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- Code metrics
- Linters
- Bug finders
- Formal verification

Linters

- Poorly written code can cause problems
- Multiple people writing code in different styles cause issues

Imagine reading this (VALID!) code…

```
public int DOSOMETHING(int num) {
  int nUmScHnIrPs = num * 2;
    int NumNirps = nUmScHnIrPs – 1;
if (NumNirps >
6) {
   if (NumNirps < 10)
   {
        return 1;
   } else
   {
     return 4;
   }
}
return 5;
}
```

Linters allow an entire team to use consistent spacing, tabs, variable naming, etc.

- Used very commonly, partly because it is so easy to use
- Any SW company worth its salt has a style guide
- Style guide can be documented (e.g. in XML) and passed to linter

Let’s see some in action…

- checkstyle – Java linter

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- Code metrics
- Linters
- Bug finders
- Formal verification

Bug Finders

- Looks for patterns that are common signs of defects
  - Many false positives, a pattern match does not mean a defect
  - Pattern DB updated continuously through open source community

- Pattern match may signal…
  - A defect
  - Confusing code that will later likely lead to defect
  - Performance issues
  - Even security vulnerabilities

Example

```
public void doStuff(int x) {
   if (x == 0) {
      x = 1;
   } else {
      x = 3;
   }
   x = 6;
}
```

Useless method

- Has no return value
- Has no side effects
- Does nothing except take up space on stack

Example

```
public static void main(String[] args) {
    double x = 0.1;
    double y = 0.2;
    double z = x + y;
    if (z == 0.3) {
        System.out.println(“math works!”);
    } else {
        System.out.println(“math is arbitrary!”);
    }
}
```

Direct Comparison of Floating-Point Values

- Floating-point values are approximations
- Always check to see if values are within epsilon of each other, e.g.
  - `if (Math.abs(z – 3.0) < 0.01) { … }`
- Or use BigDecimal, Rational, etc.

Example

```
public double calculate() {
   int x = Math.sqrt(90);
   return x;
}
```

X will always be the same value

- Just put the calculated value instead of calculating each time

Example

```
public class Quux {
   public int numBaz = 0;

   public Quux(int x) {
      numBaz = x;
   }

   public boolean equals(Object o) {
      if (o.getClass() == Quux.class) {
         return ((Quux) o).numBaz == this.numBaz;
      } else {
         return false;
      }
    }
}
```

equals() will stop working if you subclass this!

- Explicitly checking class in an equals() method
- Use this.getClass() instead

Example from a Google project

```
class MutableDouble {
  private double value_;
  public boolean equals(final Object o) {
    return o instanceof MutableDouble &&
      ((MutableDouble)o).doubleValue() == doubleValue();
  }
  public Double doubleValue() {
    return value_;
  }
```

- Can you tell where the bug is?

Example from a Google project

```
class MutableDouble {
  private double value_;
  public boolean equals(final Object o) {
    return o instanceof MutableDouble &&
      ((MutableDouble)o).doubleValue() == doubleValue();
  }
  public Double doubleValue() {
    return value_;
  }
```

- Can you tell where the bug is?

Comparison of boxed values

- double is a primitive so == operator compares numerical values
- Double is a boxed object so == compares references to objects
- `o.doubleValue() == doubleValue()`
  compares references
- Added as a pattern after discovery!

Example for Cross-site Scripting

```
public void doGet(HttpServletRequest req,
  HttpServletResponse res) {
  String target = req.getParameter("url");
  InputStream in = getResourceAsStream("META-
INF/resources/” + target);
  if (in == null) {
    res.getWriter().println("<p>Unable to locate
resource: ” + target);
  return;
  }
```

- Where is the security vulnerability?

Example for Cross-site Scripting

```
public void doGet(HttpServletRequest req,
  HttpServletResponse res) {
  String target = req.getParameter("url");
  InputStream in = getResourceAsStream("META-
INF/resources/” + target);
  if (in == null) {
    res.getWriter().println("<p>Unable to locate
resource: ” + target);
  return;
  }
```

- Where is the security vulnerability?

Display of Unsanitized user input

- Target is a user provided string
  - Can potentially contain JavaScript code that executes on website!
  - Must sanitize string before displaying

- Added as a pattern after discovery!

Let’s see some in action…

- Findbugs – bug-finding static analysis software
- Spotbugs – a successor to Findbugs