CS1632, Lecture 15: 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
  - No need to come up with test cases
  - No need to set up software / hardware to run the program
- 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
- Linters
- Bug finders
- Formal verification

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- 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

Choice of Language is Important

- Language decides effectiveness of compiler static analysis
  - The more semantic information is exposed, the more effective the analysis
  - E.g. trying to analyze assembly language code is not very effective
- Language features that help / harm compiler checks
  - Strong data types in Java:
    ```
    String x = "1";  // x is of type String
    x++;             // java compiler type error!
    ```
  - Weak data types in JavaScript:
    ```
    var x = "1";     // x is untyped
    x++;             // x == 2 (yes, not joking)
    ```
    → Exactly why TypeScript (JavaScript with typing) is gaining popularity

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- 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. Counter to see if a method was called at the beginning of method)

- Code coverage can mean different things though!

Code Coverage Example 1

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

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

- What is the code coverage?

Method Coverage

```
class Duck {
    public String quack(int x) {
        if (x > 0) {
            return "Quack!";
        } else {
            return “Negative Quack!”;
        }
    }
    public String quock() {
        return "Quock!";
    }
}
assertEquals(“Quack!”, quack(1));
assertEquals(“Negative Quack!”, quack(-4));
```

- Method coverage = 1 / 2 = 50%

Statement Coverage

```
class Duck {
    public String quack(int x) {
        if (x > 0) {
            return "Quack!";
        } else {
            return “Negative Quack!”;
        }
    }
    public String quock() {
        return "Quock!";
    }
}
assertEquals(“Quack!”, quack(1));
assertEquals(“Negative Quack!”, quack(-4));
```

- Statement coverage = 3 / 4 = 75%

Code Coverage Example 2

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

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

- What is the code coverage?

Code Coverage Example 2

```
public static int noogie(int x) {
    if (x < 10) {
        return 1;
    } else {
        if ((int) Math.sqrt(x) % 2 == 0) {
            return (x / 0);    // Defect
        } else {
            return 3;
        }
    }
}
assertEquals(1, noogie(5));
assertEquals(3, noogie(81));
assertEquals(3, noogie(9));
```

- Method coverage = 1 / 1 = 100%

Code Coverage Example 2

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

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

- Statement coverage = 4 / 5 = 80%

Coverage (usually) Means Statement Coverage

- Usually when somebody talks about code coverage …

… he/she means statement coverage

Other Kinds of Code Coverage

- Branch coverage: % of branch directions (e.g. if statement) covered
- Condition coverage: % of boolean expressions covered
- Path coverage: % of paths in method covered
- Parameter value coverage: % of (common) parameter values covered
- Entry/Exit coverage: % of method calls / returns covered
- State coverage:
  - % of states covered if program expressed as finite state machine (FSM)
  - Most accurate definition of coverage but not widely applicable:
    1) Only very simple programs have a finite number of states
    2) First, the program has to be converted to 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…

What does Code Coverage tell you?

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

// 100% (statement) coverage! WOO-HOO!
assertEquals(1, chirp(1, 1));
```

- Moments later … somebody calls chirp(1, 0)

Things Code Coverage Can’t Catch

- Defects triggered by different input values that cover the same paths
  - We saw this just now with `z / y;`
- Defects on a path that was “covered” but was never really traversed

```
int foo (int a, b) {
   int x = 2;
   if (a == 0) { x--; }
   if (b == 0) { x--; }
   return (int) 10 / x;
}
```

→ Testing `foo(0, 1)` and `foo(1, 0)` will get you 100% coverage,
But does not traverse the path of `foo(0, 0)`, the one triggering the defect

- And more!

Kinds of Static Tests

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

Coding Style is Important

- Poorly written code can cause issues
- 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 Enable A Team to Use Same Style

- 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
  - Checks on indentation
  - Checks on variable / method / class naming
  - Checks on comment formatting
  - Checks on code metrics
  - …

Linters

- Standalone
  - **CheckStyle**: Java Linter (we will use in our next exercise!)
  - CppLint: C++ Linter
  - ESLint: JavaScript Linter

- Included in your compiler
  - Javac –Xlint: -Xlint is an option in Javac that enables internal linter
  - Clang-tidy – part of the Clang C++ compiler

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- 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

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 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 boxed object so == compares references to objects
- `o.doubleValue() == doubleValue()` in `equals(Object o)` compares references not values!
  - Must change `Double doubleValue()` to `double doubleValue()`
  - That way, == operator compares double values
- Added as a pattern after discovery!

Example of 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 of 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
  - Sanitization: Removing all HTML tags that can be used to inject code

- Added as a pattern after discovery!

Bug Finder Tools

- Java
  - Findbugs: bug-finding static analysis software
  - **Spotbugs**: a successor to Findbugs
    (We will use in our next exercise!)
- C/C++
  - CppCheck: Findbugs equivalent for C/C++
  - Splint: Bug finder with focus on security vulnerabilities

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- Linters
- Bug finders
- Formal verification – Wait for Part 2!