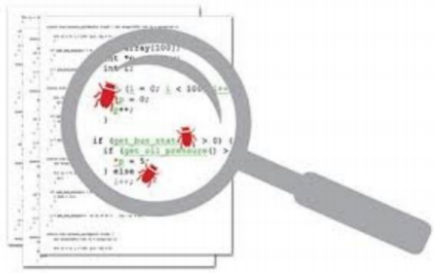
SE-Lec9

Static Analysis



What is Static Analysis?

Static analysis involves no dynamic execution of the software under test and can detect possible defects in an early stage, before running the program.

Static analysis is done after coding and before executing unit tests.

Static analysis can be done by a machine to automatically "walk through" the source code and detect noncomplying rules. The classic example is a compiler which finds lexical, syntactic and even some semantic mistakes.

Tools for Static Analysis: **Checkstyle**, **PMD**, **FindBugs**

Checkstyle

Focus on Java coding style and standards.

- whitespace and indentation (缩进)
- variable names
- Javadoc commenting
- code complexity
    - number of statements per method
    - levels of nested ifs/loops
    - lines, methods, fields, etc. per class
- proper usage
    - import statements
    - regular expressions
    - exceptions
    - I/O

- thread usage, ...

How Checkstyle works

- Checkstyle is add-ins component to IDE/Build tool
  Checkstyle 是 IDE/Build 工具的插件组件
- Coding standard is user pre-defined, and embedded in each XML file
  编码标准是用户预定义的，并嵌入在每个 XML 文件中
- Checkstyle will depend on XML file to parse code, then generate checking result to
  Checkstyle 将依赖 XML 文件解析代码，然后生成检查结果

PMD

PMD scans Java source code and looks for potential problems like:

- Possible bugs - empty try/catch/finally/switch statements
- Dead code - unused local variables, parameters and private methods
- Suboptimal code - wasteful String/StringBuffer usage
- Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
- Duplicate code - copied/pasted code means copied/pasted bugs

PMD RuleSets

- Android Rules: These rules deal with the Android SDK.
- Basic JSF rules: Rules concerning basic JSF guidelines.
- Basic JSP rules: Rules concerning basic JSP guidelines.
- Basic Rules: The Basic Ruleset contains a collection of good practices which everyone should follow.
- Braces Rules: The Braces Ruleset contains a collection of braces rules.
- Clone Implementation Rules: The Clone Implementation ruleset contains a collection of rules that find questionable usages of the clone() method.
- Code Size Rules: The Code Size Ruleset contains a collection of rules that find code size related problems.
- Controversial Rules: The Controversial Ruleset contains rules that, for whatever reason, are considered controversial.
- Coupling Rules: These are rules which find instances of high or inappropriate coupling between objects and packages.
- Design Rules: The Design Ruleset contains a collection of rules that find questionable designs.
- Import Statement Rules: These rules deal with different problems that can occur with a class' import statements.
- J2EE Rules: These are rules for J2EE
- JavaBean Rules: The JavaBeans Ruleset catches instances of bean rules not being followed.

PMD Rule Example

PMD Basic Rules

- **EmptyCatchBlock:** Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported.
- **EmptyIfStmt:** Empty If Statement finds instances where a condition is checked but nothing is done about it.
- **EmptyWhileStmt:** Empty While Statement finds all instances where a while statement does nothing. If it is a timing loop, then you should use Thread.sleep() for it; if it's a while loop that does a lot in the exit expression, rewrite it to make it clearer.
- **EmptyTryBlock:** Avoid empty try blocks - what's the point?
- **EmptyFinallyBlock:** Avoid empty finally blocks - these can be deleted.
- **EmptySwitchStatements:** Avoid empty switch statements.
- **JumbledIncrementer:** Avoid jumbled loop incrementers - it's usually a mistake, and it's confusing even if it's what's intended.
- **ForLoopShouldBeWhileLoop:** Some for loops can be simplified to while loops - this makes them more concise.

Findbugs

- Based on the concept of bug patterns (缺陷模式?). A bug pattern is a code idiom that is often an error.
- Misunderstood API methods
- FindBugs works by analyzing Java bytecode (compiled class files), so you don't even need the program's source code to use it.
- FindBugs can report false warnings, not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%
- 不关心格式或编码标准, 专注于检测潜在的错误和性能问题, 可以检测多种常见的、难以发现的错误

> 软件缺陷模式:
>
> 程序中经常出现的错误或者导致程序出现异常的缺陷，这些缺陷有同样的产生原因，同样的表示形式，按照这些缺陷发生的规律能够提取出相同的缺陷模式。
>
> ## 静态软件缺陷检测技术
>
> ### 词法分析
>
> 词法分析是对源程序进行词法分析，将其转化成单词流，然后与已定义的模式进行匹配，若匹配成功，则认为是缺陷，其原理和实现比较简单。词法分析只能检测出固定的软件缺陷，检测出的数量很有限，而且不能保证漏报率和误报率。

类型推导

类型推导是指利用机器自动推导的程序推导出程序中变量与函数的类型。主要的思想是将程序中的数据划分为不同的集合，每个集合按照一定规则，若将每个固定的集合作为一种类型，就能利用类型理论中的算法对其进行推导，其目的是保证每个推导操作都是针对合适的对象类型进行的，从而保证操作的有效性。类型推导主要适用和控制流无关的分析，能够处理大规模程序，且具有良好的扩展性，但是它不能很好的解决与控制流相关的分析。

模型检测

模型检测是由 Clarke、Emerson、Savakis、Quenelle 在 1981 年提出的，主要的思想是对软件系统构造状态机或者构造相应的抽象模型，然后在对构造后的系统进行遍历，验证系统的某种性质，是一种验证有限状态并发系统的方法。该方法能够对程序中的复杂语义进行检测，从而能够准确而高效的定位出程序中的缺陷。模型检测在很多方面得到了很好的应用，但是在软件测试中，由于软件自身具有高度的复杂性，所以该方法只能对程序某个特性构造抽象模型进行检测。

定理证明

定理证明是程序验证技术中常用的技术，是一种基于语义程序的验证技术。主要的思想是将验证求解的方法转化为数学上的定理证明，然后判断程序的抽象公式是否满足某个特定的属性。由于定理证明在处理有理数域和整数域上的运算极其的复杂，且在定理判断的过程中需要人工添加很多注释条件，不能自动的或半自动的判定缺陷程序缺陷，因而不能够得到广泛的运用。

符号执行

符号执行是一种白盒测试方法，其主要的思想是用抽象符号来表示程序中变量的值，进而模拟程序的执行，通过将程序的所有执行路径抽象成符号表达式，然后对每个特定的路径输入抽象符号，通过约束求解输出每个符号，最后判断程序的行为是否错误。按照模拟程序的所有执行路径，求得跟踪变量所有可能的值，进而发现程序中的逻辑错误。但是随着程序规模的扩大，执行路径的数目很多，不能跟踪所有路径下变量可能的值，因此不能对所有路径进行分析。

抽象解释

抽象解释理论是由 R.Cousot 和 P.Cousot 提出的一种算法，该算法通过选取逼近程序不动点语义来对程序进行分析。求解程序最小不动点一般是不可判定的，程序的输入和输出不可判定性使得无法求解程序的最小不动点。因此需要寻找到一个合适的抽象函数，作用于特定的抽象域，对某个特定的属性进行考察。虽然抽象区间域没有具体域准确，但是它减少了计算的规模和难度，增加了计算的可能性。如果找到抽象域上的最小不动点，就可以将映射后的具体域作为函数的最小不动点。

基于规则的检查

基于规则的检查是指不同的应用程序中，可能存在不同的编码规则，我们可以首先利用规则处理器来处理具体的规则，然后将其转化为程序的内部表示，最后将其用于程序的静态分析。基于规则检查的优点是对不同的系统进行分析时，可以利用相对应的规则进行检查，然后定为出程序中的软件缺陷。但是由于规则数量有限，只能分析少量特定类型的软件缺陷，不能检测大部分的缺陷。 [1]

How it works?

- Use “bug patterns” to detect potential bugs
- Examples

NullPointerException

```
Address address = client.getAddress();
if ((address != null) || (address.getPostCode() != null)) {
 ...
}
```

Uninitialized field

```
public class ShoppingCart {
   private List items;
   public addItem(Item item) {
      items.add(item);
   }
}
```

What Findbugs can do?

- FindBugs comes with over 200 rules divided into different categories:
  - *Correctness*
    E.g. infinite recursive loop, reads a field that is never written
  - *Bad practice*
    E.g. code that drops exceptions or fails to close file
  - *Performance*
  - *Multithreaded correctness*
  - *Dodgy*
    - E.g. unused local variables or unchecked casts

How to use Findbugs?

- Standalone Swing application
- Eclipse plug-in
- Integrated into the build process (Ant or)

FindBugs’ warnings

1. AT: Sequence of calls to concurrent abstraction may not be atomic
2. DC: Possible double check of field
3. DL: Synchronization on Boolean
4. DL: Synchronization on boxed primitive
5. DL: Synchronization on interned String
6. DL: Synchronization on boxed primitive values
7. Dm: Monitor wait() called on Condition
8. Dm: A thread was created using the default empty run method
9. ESync: Empty synchronized block
10. IS: Inconsistent synchronization
11. IS: Field not guarded against concurrent access
12. JLM: Synchronization performed on Lock
13. JLM: Synchronization performed on util.concurrent instance
14. JLM: Using monitor style wait methods on util.concurrent abstraction
15. LI: Incorrect lazy initialization of static field
16. LI: Incorrect lazy initialization and update of static field
17. ML: Synchronization on field in futile attempt to guard that field
18. ML: Method synchronizes on an updated field
19. MSF: Mutable servlet field
20. MWN: Mismatched notify()
21. MWN: Mismatched wait()
22. NN: Naked notify
23. NP: Synchronize and null check on the same field.

24. No: Using notify() rather than notifyAll()
25. RS: Class's readObject() method is synchronized
26. RV: Return value of putIfAbsent ignored, value passed to putIfAbsent reused
27. Ru: Invokes run on a thread (did you mean to start it instead?)
28. SC: Constructor invokes Thread.start()
29. SP: Method spins on field
30. STCAL: Call to static Calendar
31. STCAL: Call to static DateFormat
32. STCAL: Static Calendar field
33. STCAL: Static DateFormat
34. SWL: Method calls Thread.sleep() with a lock held
35. TLW: Wait with two locks held
36. UG: Unsynchronized get method, synchronized set method
37. UL: Method does not release lock on all paths
38. UL: Method does not release lock on all exception paths
39. UW: Unconditional wait
40. VO: An increment to a volatile field isn't atomic
41. VO: A volatile reference to an array doesn't treat the array elements as volatile
42. WL: Synchronization on getClass rather than class literal
43. WS: Class's writeObject() method is synchronized but nothing else is
44. Wa: Condition.await() not in loop
45. Wa: Wait not in loop