

# Lecture8 Static Analysis & Defensive Programming

---

## 1. 静态分析 Static Analysis

### 什么是静态分析

- 静态分析**不涉及被测软件的动态执行**，并且可以在运行程序之前的早期阶段检测出可能的缺陷
- 静态分析在编码之后和执行单元测试之前完成
- 静态分析可以由机器来自动“遍经”源代码，并检测不符合的规则
- 典型的例子是一个编译器，它可以发现词法、句法甚至一些语义错误

### 静态分析工具

#### Checkstyle

- <http://checkstyle.sourceforge.net/>
- Checkstyle 是一个开源开发工具，可以帮助程序员编写符合编码标准的 Java 代码
- 它自动检查 Java 代码的过程，以验证代码是否符合标准
- 这使得它对于希望执行编码标准的项目非常理想
- **关注 Java 编码风格和标准**
  - 空格和缩进
  - 变量名
  - Javadoc 注释
  - 代码完整模型
    - 每个方法的语句数
    - 嵌套if /循环的层级
    - 每个类的行、方法、字段等
  - 适当的使用
    - 导入语句
    - 正则表达式
    - 异常
    - I/O
    - 线程的使用
- **Checkstyle 是如何工作的**
  - Checkstyle 是 IDE/Build 的插件组件
  - 编码标准是用户预先定义的，并嵌入到每个XML文件中
  - 检查风格将依赖于 XML 文件来解析代码，然后生成检查结果到 XML

## PMD

- **PMD 扫描 Java 源代码并寻找潜在的问题**
  - 可能的错误 —— 空的 try/catch/finally/switch 语句
  - 死代码 —— 未使用的局部变量、参数和私有方法
  - 次优代码 —— 浪费字符串/ StringBuffer 的使用
  - 过于复杂的表达式 —— 不必要的 if 语句, for 循环
  - 重复代码 —— 复制/粘贴的代码意味着复制/粘贴的bug
- **PMD 规则集**
  - [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.
  - [JUnit Rules](#): These rules deal with different problems that can occur with JUnit tests.
  - [Jakarta Commons Logging Rules](#): Logging ruleset contains a collection of rules that find questionable usages.
  - [Java Logging Rules](#): The Java Logging ruleset contains a collection of rules that find questionable usages of the logger.
  - [Migration Rules](#): Contains rules about migrating from one JDK version to another.
  - [Migration15](#): Contains rules for migrating to JDK 1.5
  - [Naming Rules](#): The Naming Ruleset contains a collection of rules about names - too long, too short, and so forth.
  - [Optimization Rules](#): These rules deal with different optimizations that generally apply to performance best practices.
  - [Strict Exception Rules](#): These rules provide some strict guidelines about throwing and catching exceptions.
  - [String and StringBuffer Rules](#): Problems that can occur with manipulation of the class String or StringBuffer.
  - [Security Code Guidelines](#): These rules check the security guidelines from Sun.
  - [Type Resolution Rules](#): These are rules which resolve java Class files for comparisson, as opposed to a String
  - [Unused Code Rules](#): The Unused Code Ruleset contains a collection of rules that find unused code.

## Findbugs

- **什么是 FindBugs**
  - 马里兰大学一个研究项目的结果
  - 基于 bug pattern 的概念, bug pattern 是一种通常是错误的代码习惯用法
    - 困难的语言特性
    - 被误解的 API 方法
    - 在维护期间修改代码时误解的不变量

- 各种各样的错误：打字错误，布尔运算符使用错误
  - FindBugs 使用静态分析来检查 Java 字节码中出现的 bug pattern
  - 静态分析意味着 FindBugs 可以通过简单地检查程序代码来发现bug：不需要执行程序
  - FindBugs 可以报告错误的警告，而不是表示真正的错误，实际上，FindBugs 报告的错误警告率小于 50
  - 不关心格式或编码标准
  - 专注于检测潜在的 bug 和性能问题
  - 可以检测多种常见的、难以发现的 bug
- **它是如何工作的**

#### 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);
    }
}
```

- **Findbugs 可以做什么**
  - FindBugs 提供了超过 200 条划分为不同类别的规则
    - Correctness
      - 无限递归循环，读取一个从未写入的字段
    - Bad Practice
      - 删除异常或关闭文件失败的代码
    - Performance
    - Multithreaded correctness
    - Dodgy
      - 未使用的局部变量或未检查的变量
- **如何使用 Findbugs**
  - 集成到构建过程中（Ant 或 Maven）

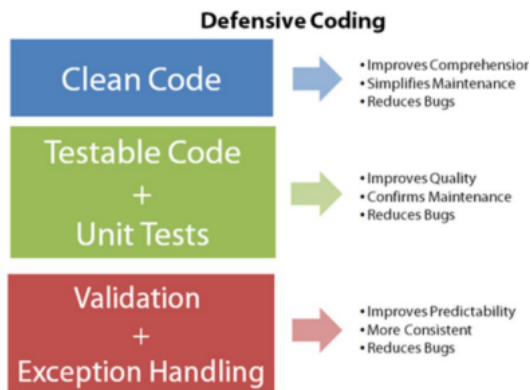
- Findbugs 警告

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

## 2. 防御式编程 Defensive Programming

确保静态分析不产生任何警告是一种防御式编程的形式

### 什么是防御式编程



- 防御性编程是一种**防御性设计**形式，旨在确保软件在不可预见的情况下继续发挥功能
- 通常在需要高可用性、安全性或保障性时使用
- 修改后加入冗余代码检查系统状态
- 隐含假设被明确地检验
- 避免了危险的编程结构
  - 指针
  - 动态存储分配
  - 浮点数
  - 并行
  - 递归
  - 中断

### Murphy 法则

如果有什么地方可以出错，那么它就会出错

### 防御式编程示例

- 使用 boolean 而不是 integer 变量
- 使用  $i \leq n$  而不是  $i == n$  条件判断
- 使用断言测试 assertion
- 在项目中构建 debug 代码，来看接口的变量
- 数据中有错误校验码 (checksum 或者 hash)

### 错误容忍 Fault Tolerance

通常的方法

- 故障探测 Failure detection
- 损失评估 Damage assessment
- 故障恢复 Fault recovery
- 故障修理 Fault repair

- N 版本编程 N-version programming
  - 并行执行独立的实现，比较结果，接受最可能的结果

## 基本技术

- 在错误后能够继续处理下一个事务
- 网络系统中的计时器和超时
- 用户中断选项（例如，强制退出，取消）
- 数据中的错误修正码
- 磁盘驱动器上的 bad block table
- 数据库中的向前和向后指针

## 向后修复 Backward Recovery

- 记录特定事件（检查点）的系统状态，失败后，重新创建最后一个检查点的状态
- 文件备份
- 将检查点与系统日志（事务审计跟踪）结合起来，允许从上一个检查点开始的事务自动重复

## 实时的软件工程 Realtime

实时计算的特殊特性要求对好的软件工程原则给予额外的关注

- 需求分析和规范
- 特殊技术（例如，锁定数据，信号量等）
- 工具开发
- 模块化设计
- 详尽测试

## 测试和调试需要特殊的工具和环境

- 调试器等不能用来测试实时性能
- 可能需要模拟环境来测试接口
  - 例如，可调时钟速度
- 通用工具可能是不可用的

## 软件开发中的安全性 Security

### 安全性目标

- 安全目标是确保与计算机系统、其数据和资源交互的代理（人或外部系统）是系统所有者希望进行此类交互的代理
- 安全性考虑需要成为整个软件开发过程的一部分，它们可能会对架构的选择产生重大影响

## 代理和组件 Agent & Component

一个大的系统将有許多代理和组件

- 两者都可能是不可靠和不安全的
- 从第三方获得的组件可能有未知的安全问题（COTS问题）

## 软件开发的挑战

- 开发安全可靠的组件
- 保护整个系统免受部分安全问题的影响

## 障碍 Barrier

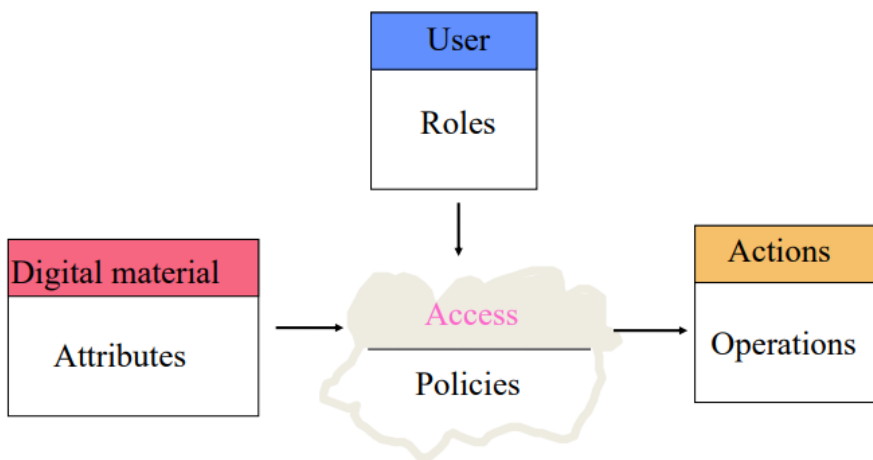
设置障碍，将复杂系统的各个部分分开

- 隔离组件，例如，不要将计算机连接到网络上
- 防火墙
- 需要身份验证才能访问某些系统或系统的某些部分

每个障碍都对系统的允许使用施加限制

- 当系统可以用简单边界划分为子系统时，障碍是最有效的

## 认证和授权 Authentication & Authorization



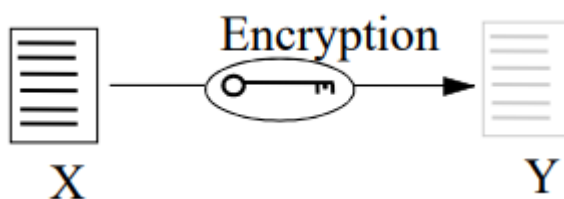
认证：建立代理的身份

- 代理知道什么（如密码）
- 代理拥有什么（如智能卡）
- 代理可以访问哪里（如控制器）

授权：确定经过身份验证的代理可以做什么

- 访问控制列表
- 小组成员

## 加密 Encryption



允许数据安全地存储和传输，即使是在这些 bit 被未经授权的代理查看时

- 私钥和公钥
- 数字签名



## 人的安全性

人们本质上是不安全的

- 粗心大意（例如，让电脑登录，使用简单的密码，把密码放在其他人可以看到的地点）
- 不诚实（例如，从金融系统偷窃）
- 恶意攻击（例如，拒绝服务攻击）

很多安全问题来自内部

- 在一个大的组织中，会有一些心怀不满和不诚实的员工
- 安全依赖于可信的个人，如果他们不诚实怎么办

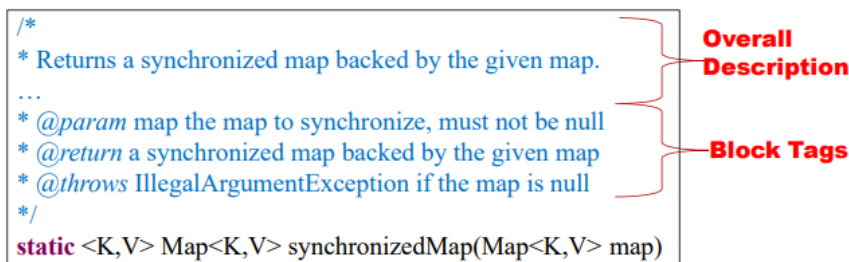
为人的不安全设计防御措施

- 让负责任的人使用这个系统
- 让不诚实或粗心的人很难接触到核心安全（如密码管理）
- 培训员工的负责任行为
- 测试系统的安全性
- 不要隐藏违规行为

## 3. 文档 Documentation

### Javadoc

#### 什么是 Javadoc 注释



```
/*
 * Returns a synchronized map backed by the given map.
 * ...
 * @param map the map to synchronize, must not be null
 * @return a synchronized map backed by the given map
 * @throws IllegalArgumentException if the map is null
 */
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
```

The diagram illustrates the structure of a Javadoc comment. The comment is enclosed in `/*` and `*/` tags. The first line after the opening tag is the overall description: `* Returns a synchronized map backed by the given map.`. This is followed by an ellipsis `...`. Then, there are three block tags: `* @param map the map to synchronize, must not be null`, `* @return a synchronized map backed by the given map`, and `* @throws IllegalArgumentException if the map is null`. The closing tag `*/` is on the next line. The code for the `synchronizedMap` method is shown below the comment. Red brackets on the right side of the diagram group the first line as the "Overall Description" and the subsequent three lines as "Block Tags".

- `@param`：参数名称，描述
- `@return`：返回值描述
- `@throws`：异常名称，抛出异常的条件

### Javadoc 格式



```

1  /**
2   * This is where the text starts. The asterisk lines
3   * up with the first asterisk above; there is a space
4   * after each asterisk. The first sentence is the most
5   * important: it becomes the “summary.”
6   *
7   * @param x Describe the first parameter (don't say its type).
8   * @param y Describe the first parameter (don't say its type).
9   * @return Tell what value is being returned (don't say its type).
10  */
11  public String myMethod(int x, int y) { // p lines up with the / in /**

```

- javadoc 注释由 `/**` 开始, `*/` 结束
  - 在javadoc注释中, 行开头的 `*` 不是注释文本的一部分
- javadoc 注释必须紧跟在以下代码的前面
  - 类 (标准类、内部类、抽象类、枚举类)
  - 接口
  - 构造器
  - 方法
  - 字段 (实例/静态)
- 在其他任何地方, javadoc 注释将被忽略

## Javadoc 是为谁准备的

- Javadoc 注释应该是为那些想要使用它的程序员编写的
- Javadoc 注释不是为
  - 需要调试、维护或升级代码的程序员
  - 只是想使用这个程序的人
- 因此
  - Javadoc 注释应该准确地描述如何使用类、方法、构造函数等
  - Javadoc 注释不应该描述类或方法的内部工作 (除非它影响到用户)
  - 此外, Javadoc 方法注释不应该告诉谁使用了方法 (不合适, 但也很难保持更新)

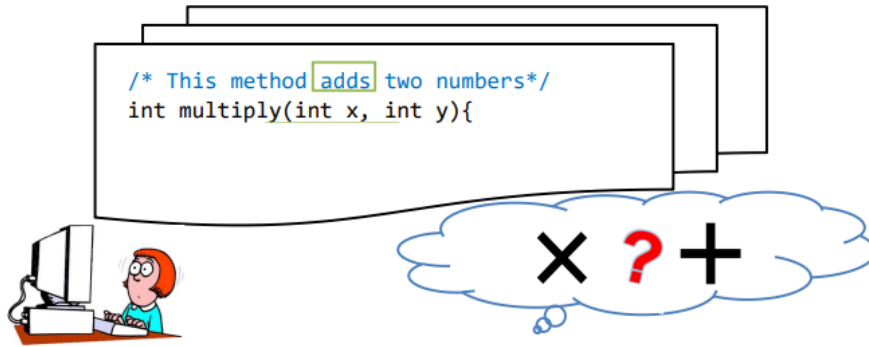
## Javadoc 注释中的 Tag

使用 Javadoc Tag 中的标准的顺序

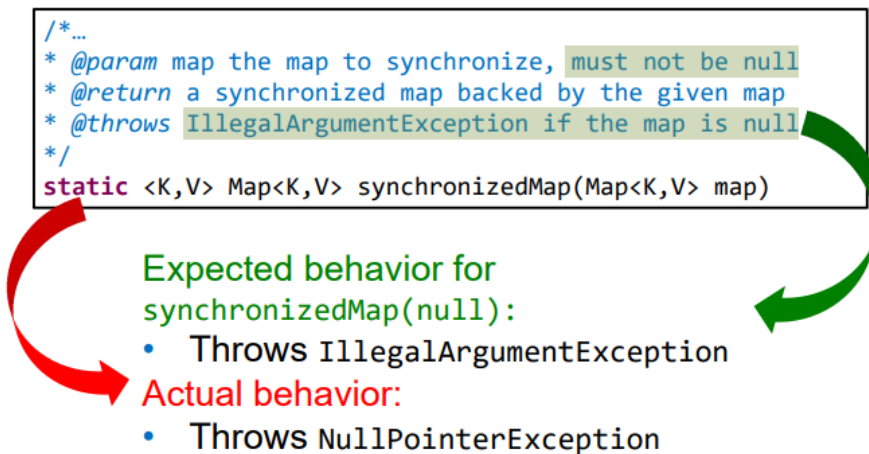
- 在类和接口的描述中, 使用
  - `@author` 你的名字
    - 在所有顶级类的作业中使用 `@author`
  - `@version` 版本名称或者日期
- 在方法的描述中, 使用
  - `@param` p 一个对参数 p 的描述
  - `@return` 返回值的描述 (如果方法返回 void 则不使用)
  - `@exception` e 描述任何抛出的异常

## Javadoc 存在的问题

- 过时的代码注释

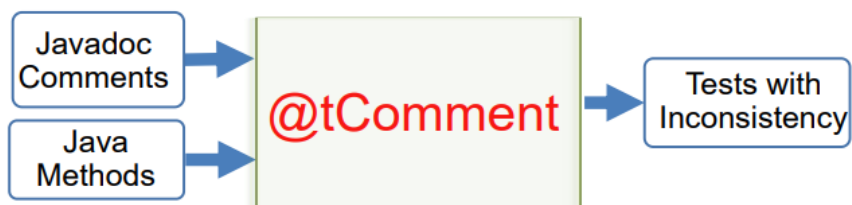


- Javadoc 注释与代码不一致

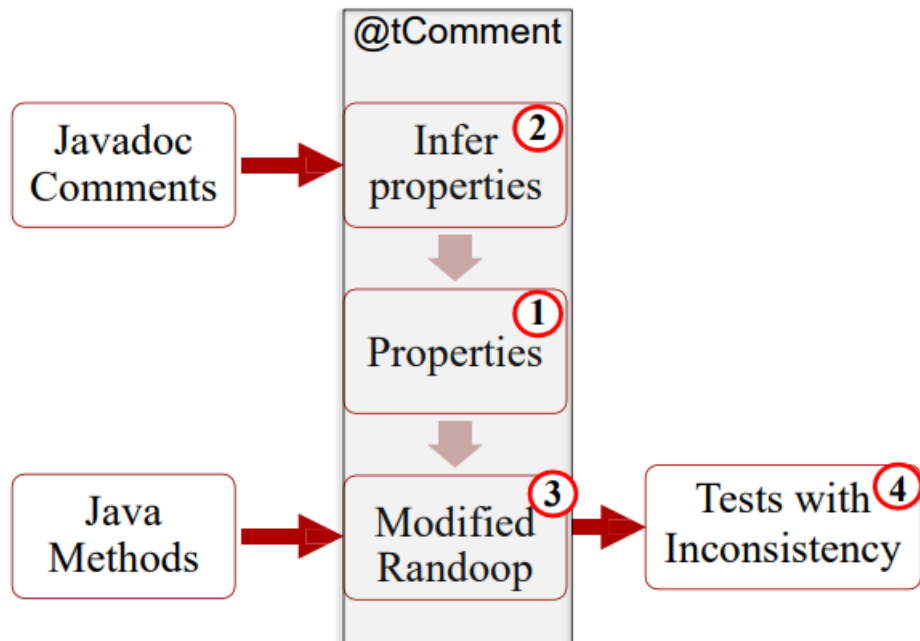


- 正确的代码，不正确的注释
- 错误的代码正确的注释

## 检查注释和代码的一致性的方法



# @tComment Design



## Bug 和暂时没有添加的功能

- 写下已知的问题
  - 有三个“标准”标志，您可以将它们放入任何公共标记中
    - TODO：描述应该添加的功能
    - FIXME：描述方法中的 bug
    - XXX：需要更多的考虑
- 可以在 IDE 中创建自己的附加标记