哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 7: Software Construction for Robustness

7.3 Assertions and Defensive Programming

Wang Zhongjie
rainy@hit.edu.cn

May 6, 2018

Outline

- **Recall: Designing an ADT**
- **Assertions**
  - What to Assert and What not to?
  - Guidelines for Using Assertions
- **Defensive Programming**
  - Techniques for defensive programming
- **The FindBugs tool**
- **Summary**

7-2节：使用错误处理和exception提高correctness和robustness，但主要还是面向robustness。

本节：第2种技术——断言、防御式编程，侧重于correctness。

Reading

- **MIT 6.031：08**
- **代码大全：第8章**

1 Recall: Designing an ADT

First Defense: Make Bugs Impossible

- **The best defense against bugs is to make them impossible by design.** 最好的防御就是不要引入bug ☺
  - **Static checking**: eliminates many bugs by catching them at compile time.
  - **Dynamic checking**: Java makes array overflow bugs impossible by catching them dynamically. If you try to use an index outside the bounds of an array or a `List`, then Java automatically produces an error. --- unchecked exception / runtime error
  - **Immutability** (immunity from change): An immutable type is a type whose values can never change once they have been created.
  - **Immutable references**: by `final`

Second Defense: Localize Bugs

- If we can't prevent bugs, we can **try to localize them to a small part of the program**, so that we don't have to look too hard to find the cause of a bug. 如果无法避免，尝试着将bug限制在最小的范围内
  - When localized to a single method or small module, bugs may be found simply by studying the program text. 限定在一个方法内部，不扩散
  - **Fail fast**: the earlier a problem is observed (the closer to its cause), the easier it is to fix. 尽快失败，就容易发现、越早修复

```
/**
 * @param x  requires x >= 0
 * @return approximation to square root of x
 */
public double sqrt(double x) {

    ...
}
```

Pre-condition
如果违反，该方法可以做任何事

Since the bad call indicates a bug in the caller, however, the most useful behavior would point out the bug as early as possible. 应该尽可能早的指出client的bug

Second Defense: Localize Bugs

- **Assertions 断言:** When the precondition is not satisfied, this code terminates the program by throwing an `AssertionError` exception. The effects of the caller's bug are prevented from propagating. Fail fast，避免扩散

- **Checking preconditions is an example of defensive programming**
  检查前置条件是防御式编程的一种典型形式
  - Real programs are rarely bug-free.
  - Defensive programming offers a way to mitigate the effects of bugs even if you don't know where they are.

2 Assertions

(1) What and Why Assertions?

What is assertion?

- **An assertion is code that's used during development that allows a program to check itself as it runs, i.e.,** to *test your assumptions* about your program logic (such as pre-conditions, post-conditions, and invariants). 断言：在开发阶段的代码中嵌入，检验某些“假设”是否成立。若成立，表明程序运行正常，否则表明存在错误。
  - When an assertion is true, that means everything is operating as expected.
  - When it's false, that means it has detected an unexpected error in the code.
- **An example**

```
public class AssertionTest {
   public static void main(String[] args) {
      int number = -5;    // assumed number is not negative
      // This assert also serve as documentation
      assert (number >= 0) : "number is negative: " + number;
      // do something
      System.out.println("The number is " + number);
   }
}
```

What is assertion?

- **Each assertion contains a `boolean` expression that you believe will be true when the program executes.**
  - If it is not true, the JVM will throw an `AssertionError`.
  - This error signals you that you have an invalid assumption that needs to be fixed. 出现`AssertionError`，意味着内部某些假设被违反了
  - The assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors. 增强程序员对代码质量的信心：对代码所做的假设都保持正确

- **Assertion is much better than using `if-else` statements, as it serves as proper documentation on your assumptions, and it does not carry performance liability in the production environment.**
  断言即是对代码中程序员所做假设的文档化，也不会影响运行时性能(在实际使用时，assertion都会被disabled)

What is assertion?

- **An assertion usually takes two arguments**
  - **A boolean expression** that describes the assumption supposed to be true
  - **A message** to display if it isn't.
- **The Java language has a keyword `assert` with two forms:**
  - `assert condition;`
  - `assert condition : message;`
  - Both statements evaluate the condition and throw an `AssertionError` if the boolean expression evaluates to false.
  - In the second statement, the expression is passed to the constructor of the `AssertionError` object and turned into a message string. The description is printed in an error message when the assertion fails, so it can be used to provide additional details to the programmer about the cause of the failure.
    所构造的message在发生错误时显示给用户，便于快速发现错误所在

Assertion in Java

- To assert that `x` is non-negative, you can simply use the statement

```
assert x >= 0;
```

- Or pass the actual value of `x` into the `AssertionError` object, so that it gets displayed later:

```
assert x >= 0 : “x is ” + x;
```

- If `x == -1`, then this assertion fails with the error message

```
x is -1
```

- This information is often enough to get started in finding the bug.

Assertion Example

```
public class AssertionSwitchTest {
   public static void main(String[] args) {
      // assumed either '+', '-', '*', '/' only
      char operator = '%';
      int operand1 = 5, operand2 = 6, result = 0;
      switch (operator) {
         case '+': result = operand1 + operand2; break;
         case '-': result = operand1 - operand2; break;
         case '*': result = operand1 * operand2; break;
         case '/': result = operand1 / operand2; break;
         default: assert false : "Unknown operator: " + operator;
      }
      System.out.println(operand1 + " " + operator + " "
               + operand2 + " = " + result);
   }
}
```

Why use assertions?

- **Document & test programmer's assumptions, e.g., invariants;**
- **Verify programmer's understanding**
- **Quickly uncover bugs**
- **Increase confidence that program is bug-free**
- **Asserts turn black box tests into white box tests**

Experience has shown that writing assertions while programming is one of the ***quickest and most effective*** ways to detect and correct bugs.

As an added benefit, assertions serve to document the inner workings of your program, ***enhancing maintainability***.

(2) What to Assert and What not to?

When use assertions?

- **Assertion can be used for verifying:**
  - **Internal Invariants 内部不变量**: Assert that a value is within a certain constraint, e.g., `assert x > 0`.
  - **Rep Invariants 表示不变量**: Assert that an object's state is within a constraint. What must be true about each instance of a class before or after the execution of a method? Class invariants are typically verified via private `boolean` method, e.g., `checkRep()`.
  - **Control-Flow Invariants 控制流不变量**: Assert that a certain location will not be reached. For example, the default clause of a `switch-case` statement.
  - **Pre-conditions of methods 方法的前置条件**: What must be true when a method is invoked? Typically expressed in terms of the method's arguments or the states of its objects.
  - **Post-conditions of methods 方法的后置条件**: What must be true after a method completes successfully?

What to Assert?

- **Pre-condition: method argument requirements**
- **Post-condition: Method return value requirements**
  - This kind of assertion is sometimes called a ***self check*** .

```
public double sqrt(double x) {
    assert x >= 0;
    double r;
    ... // compute result r
    assert Math.abs(r*r - x) < .0001;
    return r;
}
```

What to Assert?

- **Control-flow: covering all**
  - If a conditional statement or switch does not cover all the possible cases, it is good practice to use an assertion to block the illegal cases.

```
void foo() {
    for (...) {
      if (...)
        return;
    }
    assert false;
    // Execution should never reach this point!
}
```

```
switch (vowel) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u': return "A";
      default: assert false;
}
```

What to Assert: more scenarios

- **A file or stream is open (or closed) when a method begins executing (or when it ends executing)**
- **A file or stream is at the beginning (or end) when a method begins executing (or when it ends executing)**
- **A file or stream is open for read-only, write-only, or both read and write**
- **The value of an input-only variable is not changed by a method**

What to Assert: more scenarios

- **A pointer is non-NULL**
- **An array or other container passed into a method can contain at least X number of data elements**
- **A table has been initialized to contain real values**
- **A container is empty (or full) when a method begins executing (or when it finishes)**
- **The results from a highly optimized, complicated method match the results from a slower but clearly written routine**

Example

```java
/**
 * Solves quadratic equation ax^2 + bx + c = 0.
 *
 * @param a quadratic coefficient, requires a != 0
 * @param b linear coefficient
 * @param c constant term
 * @return a list of the real roots of the equation
 */
public static List<Double> quadraticRoots(final int a, final int b, final int c) {
    List<Double> roots = new ArrayList<Double>();
    // A
    ... // compute roots
    // B
    return roots;
}
```

What statements would be reasonable to write at position A?

And at position B?

- `assert a != 0;`
- `assert b != 0;`
- `assert c != 0;`
- `assert roots.size() >= 0;`
- `assert roots.size() <= 2;`
- `for (double x : roots) { assert Math.abs(a*x*x + b*x + c) < 0.0001; }`

When to use assertions?

- **Normally, you don't want users to see assertion messages in production code; assertions are primarily for use during development and maintenance.** 断言主要用于开发阶段，避免引入和帮助发现bug
- **Assertions are normally compiled into the code at development time and compiled out of the code for production.** 实际运行阶段，不再使用断言
- **During production, they are compiled out of the code so that the assertions don't degrade system performance.** 避免降低性能
- **When should you write runtime assertions?** 使用断言的主要目的是为了在开发阶段调试程序、尽快避免错误
  - As you write the code, not after the fact. When you're writing the code, you have the invariants in mind.
  - If you postpone writing assertions, you're less likely to do it, and you're liable to omit some important invariants.

What Not to Assert?

- **Runtime assertions are not free. They can clutter the code, so they must be used judiciously.**
  - Avoid trivial assertions, just as you would avoid uninformative comments.

```
// don't do this:
x = y + 1;
assert x == y+1;
```

  - This assertion doesn't find bugs in your code.
  - It finds bugs in the compiler or Java virtual machine, which are components that you should trust until you have good reason to doubt them.
  - If an assertion is obvious from its local context, leave it out.

Avoid putting executable code in assertions

- **Since assertions may be disabled, the correctness of your program should never depend on whether or not the assertion expressions are executed.**
- **In particular, asserted expressions should not have *side-effects*.**
  - For example, if you want to assert that an element removed from a list was actually found in the list, don't write it like this:
- **If assertions are disabled, the entire expression is skipped, and x is never removed from the list. Write it like this instead:**

```
// don't do this:
assert list.remove(x);
```

```
// do this:
boolean found = list.remove(x);
assert found;
```

Don't Assert External Conditions

- **Never use assertions to test conditions that are external to your program. 程序之外的事，不受你控制，不要乱断言**
  - Such as the existence of files, the availability of the network, or the correctness of input typed by a human user. 文件/网络/用户输入等
  - Assertions test the internal state of your program to ensure that it is within the bounds of its specification. 断言只是检查程序的内部状态是否符合规约
  - When an assertion fails, it indicates that the program has run off the rails in some sense, into a state in which it was not designed to function properly. Assertion failures therefore indicate bugs. 断言一旦false，程序就停止执行
  - External failures are not bugs, and there is no change you can make to your program in advance that will prevent them from happening. 你的代码无法保证不出现此类外部错误 (recall section 7-2)
  - External failures should be handled using exceptions instead. Avoid trivial assertions, just as you would avoid uninformative comments. 外部错误要使用Exception机制去处理

Turn on/off Assert in different phases

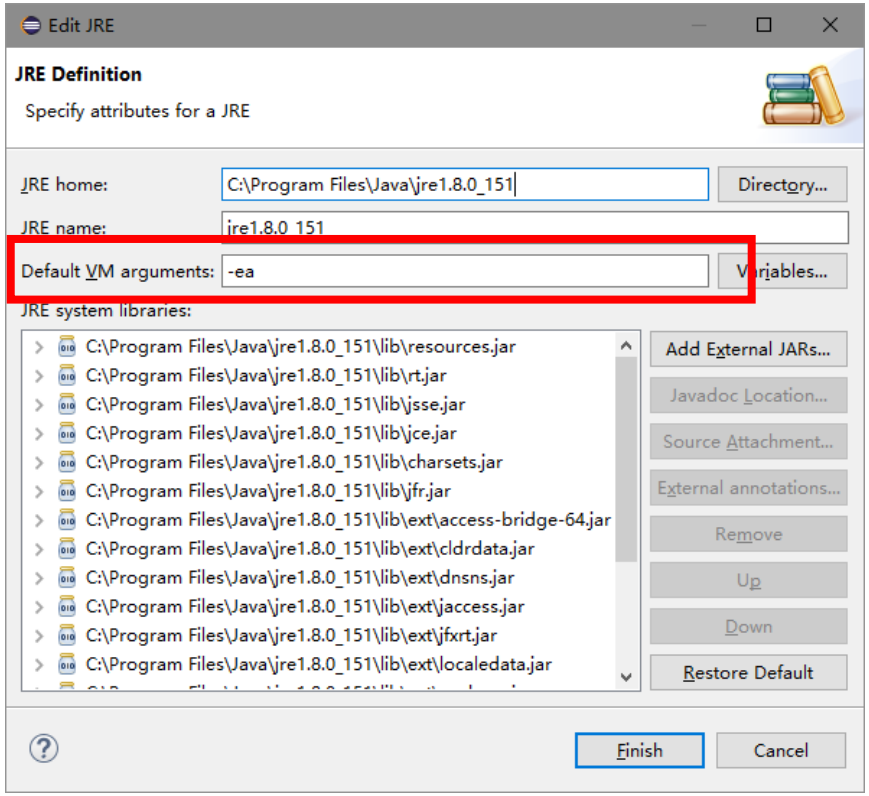
- **Many assertion mechanisms are designed so that assertions are executed only during testing and debugging, and turned off when the program is released to users.**
  - Assertions are a great tool for keeping your code safe from bugs, but Java has them off by default! Java缺省关闭断言，要记得打开(`-ea`)
- **The advantage of this approach is that you can write very expensive assertions that would otherwise seriously degrade the performance of your program.** 断言非常影响运行时的性能
  - For example, a procedure that searches an array using binary search has a requirement that the array be sorted.
  - Asserting this requirement requires scanning through the entire array, however, turning an operation that should run in logarithmic time into one that takes linear time.
  - You should be willing to pay this cost during testing, since it makes debugging much easier, but not after the program is released to users.

Suggestion: always enable assertions

- **However, disabling assertions in release has a serious disadvantage.**
- **With assertions disabled, a program has far less error checking when it needs it most.**
- **Novice programmers are usually much more concerned about the performance impact of assertions than they should be.**
- **Most assertions are cheap, so they should not be disabled in the official release.**

Enable & Disable assertions in Java

- **Enable assertions**
  - Running the program with the `-enableassertions` or `-ea` option:
    - `java -enableassertions MyApp`
  - The option -ea... turns on assertions in all classes of the default package.
    - `java -ea:MyClass MyApp`
- **Disable assertions**
  - Running the program with the `-disableassertions` or `-da` option:
    - `java -ea:... -da:MyClass MyApp`
- **By default, assertions are disabled.**
- **Enable assertions in Eclipse:**
  - In preferences, go to Java → Installed JREs . Click "Java SE 8", click "Edit…", and in the "Default VM arguments" box enter: `-ea`



`assert` vs. `assertXXX()`... in JUnit

- **The Java assert statement is a different mechanism from the JUnit methods `assertTrue()`, `assertEquals()`, etc.**
- **They all assert a predicate about your code, but are designed for use in different contexts.**
- **The `assert` statement should be used in implementation code, for defensive checks inside the implementation.**
- **JUnit `assertXXX()` methods should be used in JUnit tests, to check the result of a test.**
- **The `assert` statements don't run without `-ea`, but the JUnit `assertXXX()` methods always run.**

(3) Guidelines for Using Assertions

Assertion vs. Exception?

- **Assertions generally cover *correctness* issues of program.**
  - If an assertion is fired for an anomalous condition, the corrective action is not merely to handle an error gracefully—the corrective action is to change the program's source code, recompile, and release a new version of the software. 断言→Correctness
- **Exceptions generally cover *robustness* issues of program.**
  - If error handling code is used to address an anomalous condition, the error handling will enable the program to respond to the error gracefully. 错误/异常处理→Robustness
- **Assertions are especially useful in large, complicated programs and in high reliability programs.**
  - They enable programmers to more quickly flush out mismatched interface assumptions, errors that creep in when code is modified, and so on.

Assertion vs. Exception?

- **Use error handling code (exception) for conditions you expect to occur 使用异常来处理你“预料到可以发生”的不正常情况**
  - Error handling code checks for off-nominal circumstances that might not occur very often, but that have been anticipated by the programmer who wrote the code and that need to be handled by the production code.

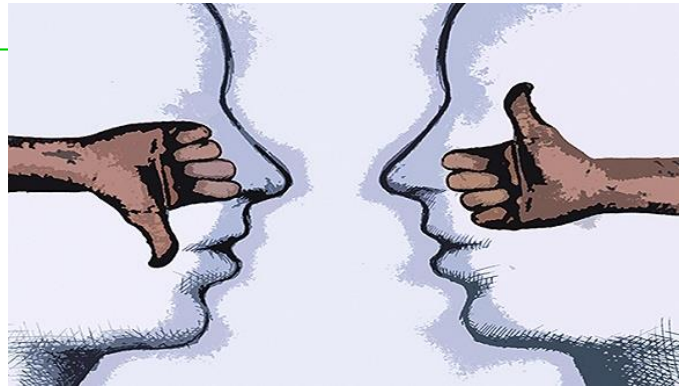
- **Use assertions for conditions that should never occur 使用断言处理“绝不应该发生”的情况**
  - Assertions check for bugs in the code.

Use Assertions for pre-/post- conditions

- **Preconditions and postconditions are part of an approach to program design and development known as "design by contract".**
  - When preconditions and postconditions are used, each routine or class forms a contract with the rest of the program.
  - Preconditions are the client code's obligations to the code it calls.
  - Postconditions are the routine or class's obligations to the code that uses it.

- **Assertions are a useful tool for documenting preconditions and postconditions.**
  - Comments could be used to document preconditions and postconditions, but assertions can check dynamically whether the preconditions and postconditions are true.

Should pre-/post-condition be asserted?



- **Another viewpoint: Do not use assertions for argument checking in public methods.** 在其他一些开发者眼里，不应该针对参数的合法性使用断言。
- **Reasons:**
  - Argument checking is typically part of the published specifications (or contract) of a method, and these specifications must be obeyed whether assertions are enabled or disabled. 不管是否-ea，spec中的pre-/post-conditions都能够被保证
  - Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (such as `IllegalArgumentException`, `IndexOutOfBoundsException`, or `NullPointerException`). An assertion failure will not throw an appropriate exception. 即使spec被违反，也不应通过assert直接fail，而是应抛出具体的runtime异常

Use Assertions for pre-/post- conditions

- If the variables `latitude`, `longitude`, and `elevation` were coming from an external source, invalid values should be checked and handled by error handling code rather than assertions. 如果参数来自于外部（不受自己控制），使用异常处理
- If the variables are coming from a trusted, internal source, however, and the routine's design is based on the assumption that these values will be within their valid ranges, then assertions are appropriate. 如果来自于自己所写的其他代码，可以使用断言来帮助发现错误（例如post-condition就需要）

```
float latitude;
float longitude;
float elevation;

//Preconditions
assert latitude>=-90 && latitude<=90;
assert longitude>=0 && longitude<360;
assert elevation>=-500 && elevation<=7500
//Postconditions
assert Velocity>=0 && Velocity <= 600

return Velocity;
```

Should pre-/post-condition be asserted?

```
/**
  * Sets the refresh rate.
  *
  * @param  rate refresh rate, in frames per second.
  * @throws IllegalArgumentException if rate <= 0 or
  * rate > MAX_REFRESH_RATE.
*/

public void setRefreshRate(int rate) {
  // Enforce specified precondition in public method
  if (rate <= 0 || rate > MAX_REFRESH_RATE)
    throw new IllegalArgumentException("Illegal rate: " + rate);

  setRefreshInterval(1000/rate);
}
```

Should pre-/post-condition be asserted?

- **You can use an assertion to test a nonpublic method's precondition that you believe will be true no matter what a client does with the class.**

```
/**
 * Sets the refresh interval (which must correspond to a legal rate).
 *
 * @param  interval refresh interval in milliseconds.
*/

 private void setRefreshInterval(int interval) {
  // Confirm adherence to precondition in nonpublic method
  assert interval > 0
         && interval <= 1000/MAX_REFRESH_RATE : interval;

  ... // Set the refresh interval
 }
```

Should pre-/post-condition be asserted?

- **You can test postcondition with assertions in both public and nonpublic methods.**

```
/**
  * Returns a BigInteger whose value is (this-1 mod m).
  *
  * @param  m the modulus.
  * @return this-1 mod m.
  * @throws ArithmeticException  m <= 0, or this BigInteger
  *  has no multiplicative inverse mod m (that is, this BigInteger
  *  is not relatively prime to m).
  */
public BigInteger modInverse(BigInteger m) {
  if (m.signum <= 0)
    throw new ArithmeticException("Modulus not positive: " + m);
  ... // Do the computation
  assert this.multiply(result).mod(m).equals(ONE) : this;
  return result;
}
```

Combine assert & exception handling for robustness

- **Both assertions and exception handling code might be used to address the same error.** **断言和异常处理都可以处理同样的错误**
  - In the source code for Microsoft Word, for example, conditions that should always be true are asserted, but such errors are also handled by error-handling code in case the assertion fails.
  - For extremely large, complex, long-lived applications like Word, assertions are valuable because they help to flush out as many development-time errors as possible.
- **But the application is so complex (million of lines of code) and has gone through so many generations of modification that it isn't realistic to assume that every conceivable error will be detected and corrected before the software ships, and so errors must be handled in the production version of the system as well.**

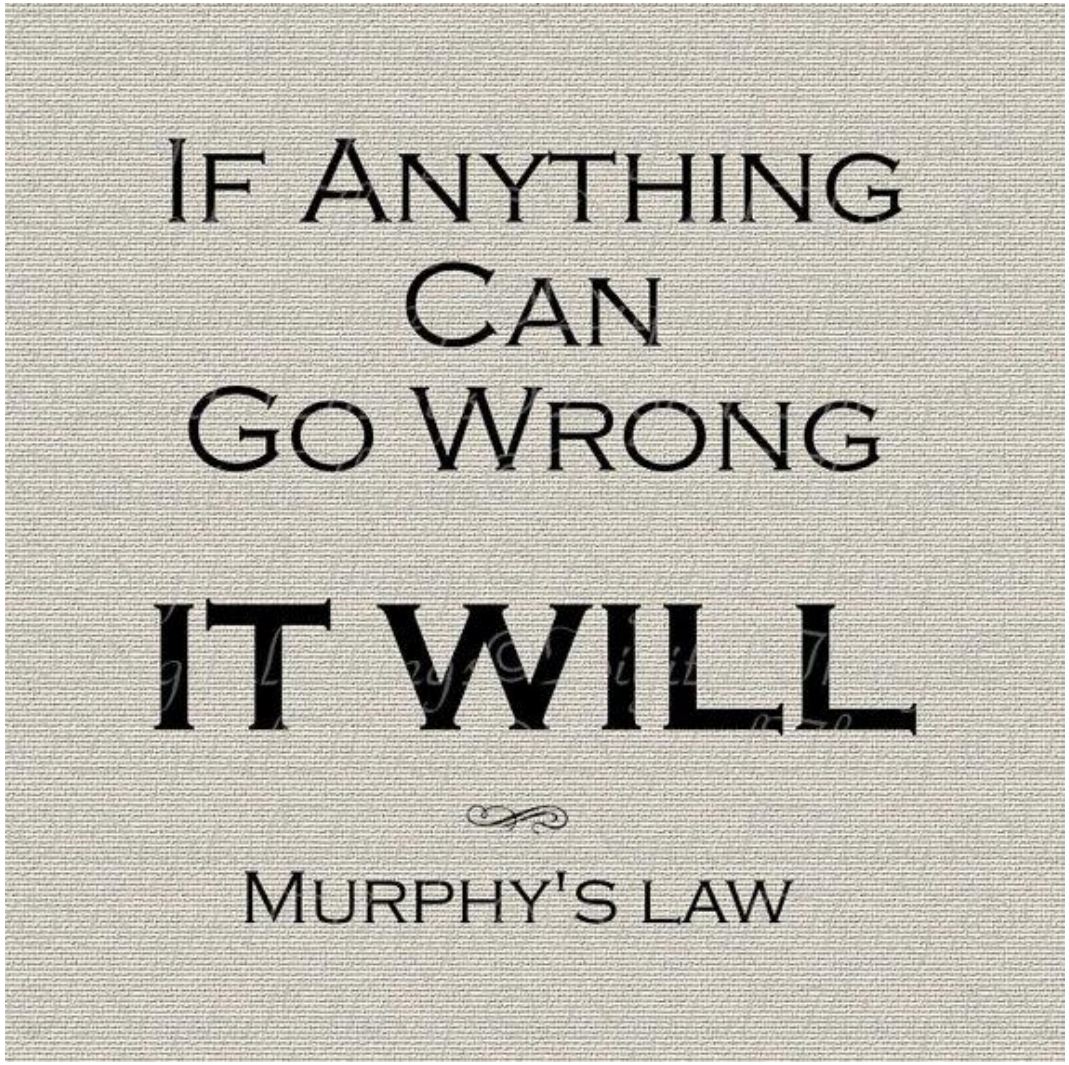
开发阶段用断言尽可能消除bugs
在发行版本里用异常处理机制处理漏掉的错误

3 Defensive Programming

What is defensive programming?

- **Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances.**
  - Defensive programming practices are often used where high availability, safety or security is needed.
- **The idea can be viewed as reducing or eliminating the prospect of Murphy's Law having effect.**
- **"Your code should fail as early as possible!"**

Murphy's Law



What is defensive programming?

- The idea is based on **defensive driving**
  - You adopt the mind-set that you're never sure what the other drivers are going to do.
  - That way, you make sure that if they do something dangerous you won't be hurt.
  - You take responsibility for protecting yourself even when it might be the other driver's fault.
  - 眼观六路，耳听八方，一旦其他车辆有对你产生危险的症状，马上采取防御式行动

Benefits of defensive programming

- **Defensive programming is an approach to improve software and source code, in terms of:**
  - **General quality –** reducing the number of software bugs and problems.
  - **Making the source code comprehensible –** the source code should be readable and understandable so it is approved in a code audit.
  - **Making the software behave in a predictable manner** despite unexpected inputs or user actions.

Techniques for defensive programming

Techniques for defensive programming

- **Protecting programs from invalid inputs**
- **Assertions**
- **Exceptions**
- **Specific error handling techniques**
- **Barricade**
- **Debugging aids**

- **The best form of defensive coding is not inserting errors in the first place.**
- **You can use defensive programming in combination with the other techniques.**

(1) Protecting Programs From Invalid Inputs

- **"Garbage in, garbage out"**
  - That expression is essentially software development's version of caveat emptor: let the user beware. 货物出门概不退换
- **For production software, garbage in, garbage out isn't good enough.**
- **A good program never puts out garbage, regardless of what it takes in.**
  - "Garbage in, nothing out"
  - "Garbage in, error message out"
  - "No garbage allowed in"

- **"Garbage in, garbage out" is the mark of a sloppy, non-secure program.**

Protecting Programs From Invalid Inputs

- **Check the values of all data from external sources 对来自外部的数据源要仔细检查，例如：文件、网络数据、用户输入等**
  - When getting data from a file, a user, the network, or some other external interface, check to be sure that the data falls within the allowable range.
- **Examples:**
  - Make sure that numeric values are within tolerances and that strings are short enough to handle.
  - If a string is intended to represent a restricted range of values (such as a financial transaction ID or something similar), be sure that the string is valid for its intended purpose; otherwise reject it.
  - If you're working on a secure application, be especially leery of data that might attack your system: attempted buffer overflows, injected SQL commands, injected html or XML code, integer overflows, and so on.

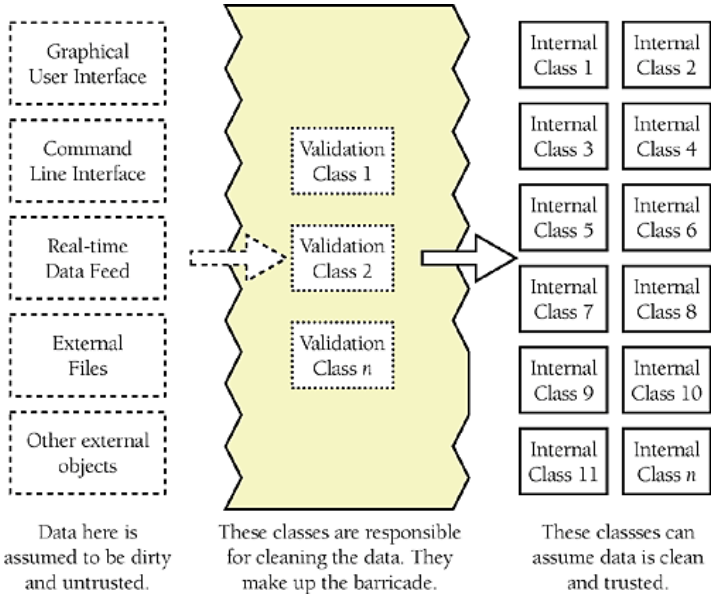
Protecting Programs From Invalid Inputs

- **Check the values of all routine input parameters** 对每个函数的输入参数合法性要做仔细检查，并决定如何处理非法输入
  - Checking the values of routine input parameters is essentially the same as checking data that comes from an external source, except that the data comes from another routine instead of from an external interface.

- **Decide how to handle bad inputs**
  - Once you've detected an invalid parameter, what do you do with it?
  - Depending on the situation, you might choose any of a dozen different approaches, which are described in detail later in this chapter.

(2) Barricade 设置路障

- **Barricades are a damage-containment strategy.**
  - The reason is similar to that for having isolated compartments in the hull of a ship and firewalls in a building.
  - One way to barricade for defensive programming purposes is to designate certain interfaces as boundaries to "safe" areas.
  - Check data crossing the boundaries of a safe area for validity and respond sensibly if the data isn't valid.



Defining some parts of the software that work with dirty data and some that work with clean can be an effective way to relieve the majority of the code of the responsibility for checking for bad data.

Barricade

- **The class's public methods assume the data is unsafe, and they are responsible for checking the data and sanitizing it. 类的public方法接收到的外部数据都应被认为是dirty的，需要处理干净再传递到private方法——隔离舱**
  - Once the data has been accepted by the class's public methods, the class's private methods can assume the data is safe.
- **Another way is as an operating-room technique. 操作间技术**
  - Data is sterilized before it's allowed to enter the operating room. Anything that's in the operating room is assumed to be safe.
  - The key design decision is deciding what to put in the operating room, what to keep out, and where to put the doors—which routines are considered to be inside the safety zone, which are outside, and which sanitize the data.
  - The easiest way to do this is usually by sanitizing external data as it arrives, but data often needs to be sanitized at more than one level, so multiple levels of sterilization are sometimes required.

Barricade

- **Convert input data to the proper type at input time**
  - Input typically arrives in the form of a string or number.
  - Sometimes the value will map onto a boolean type like “yes” or “no.”
  - Sometimes the value will map onto an enumerated type like Color_Red, Color_Green, and Color_Blue.
  - Carrying data of questionable type for any length of time in a program increases complexity and increases the chance that someone can crash your program by inputting a color like “Yes.”
  - Convert input data to the proper form as soon as possible after it’s input.

Relationship between Barricades and Assertions

- **The use of barricades makes the distinction between assertions and error handling clean cut.**
  - Routines that are outside the barricade should use error handling because it isn't safe to make any assumptions about the data.
  - Routines inside the barricade should use assertions, because the data passed to them is supposed to be sanitized before it's passed across the barricade. If one of the routines inside the barricade detects bad data, that's an error in the program rather than an error in the data.
- **“隔离舱”外部的函数应使用异常处理，“隔离舱”内的函数应使用断言。**
  - The use of barricades also illustrates the value of deciding at the architectural level how to handle errors.
  - Deciding which code is inside and which is outside the barricade is an architecture-level decision.
- **Proxy设计模式？——隔离**

(3) Debugging Aids

- **Debugging aids is a key aspect of defensive programming for quickly detecting errors. (See 7.4 debugging)**
- **Don't Automatically Apply Production Constraints to the Development Version.**
  - A common programmer blind spot is the assumption that limitations of the production software apply to the development version.

Production version	Development version
has to run fast	might be able to run slow
has to be stingy with resources	might be allowed to use resources extravagantly
shouldn't expose dangerous operations to the user	can have extra operations that you can use without a safety net

  - Be willing to trade speed and resource usage during development in exchange for built-in tools that can make development go more smoothly.

Debugging Aids

- **Introduce Debugging Aids Early.**
  - The earlier you introduce debugging aids, the more they'll help.
  - Typically, you won't go to the effort of writing a debugging aid until after you've been bitten by a problem several times.
  - If you write the aid after the first time, however, or use one from a previous project, it will help throughout the project.

Debugging Aids: Use Offensive Programming

- **Use Offensive Programming**
  - Exceptional cases should be handled in a way that makes them obvious during development and recoverable when production code is running.
  - "Offensive programming" (Howard and LeBlanc 2003).
  - Suppose you have a case statement that you expect to handle only five kinds of events.
  - During development, the default case should be used to generate a warning that says "Hey! There's another case here! Fix the program!"
  - During production, however, the default case should do something more graceful, like writing a message to an error-log file.

Debugging Aids: Use Offensive Programming

- **Some ways you can program offensively:**
  - Make sure asserts abort the program. Don't allow programmers to get into the habit of just hitting the ENTER key to bypass a known problem. Make the problem painful enough that it will be fixed.
  - Completely fill any memory allocated so that you can detect memory allocation errors.
  - Completely fill any files or streams allocated to flush out any file-format errors.
  - Be sure the code in each case statement's else clause fails hard (aborts the program) or is otherwise impossible to overlook.
  - Fill an object with junk data just before it's deleted
- **Sometimes the best defense is a good offense.**
- **Fail hard during development so that you can fail softer during production.**

Debugging Aids: Plan to Remove Debugging Aids

- **In commercial software, leaving all the debugging code in the program will cause that the performance penalty in size and speed can be prohibitive.**
- **Plan to avoid shuffling debugging code in and out of a program.**
  - Use version control and build tools like make
    - Version-control tools(like ant, make…) can build different versions of a program from the same source files.
    - In development mode, you can set the build tool to include all the debug code.
    - In production mode, you can set it to exclude any debug code you don't want in the commercial version.
  - Use a built-in preprocessor
    - If your programming environment has a preprocessor, you can include or exclude debug code at the flick of a compiler switch.
    - You can use the preprocessor directly or by writing a macro that works with preprocessor definitions.

Debugging Aids: Use debugging stubs

- **In many instances, you can call a routine to do debugging checks.**
- **During development, the routine might perform several operations before control returns to the caller.**
- **For production code, you can replace the complicated routine with a stub routine that merely returns control immediately to the caller or performs only a couple of quick operations before returning control.**
- **This approach incurs only a small performance penalty, and it's a quicker solution than writing your own preprocessor.**
- **Keep both the development and production versions of the routines so that you can switch back and forth during future development and production.**

(1) Leave in code that checks for important errors

- **One of the paradoxes of defensive programming is that during development, you'd like an error to be noticeable.**
- **But during production, you'd rather have the error be as unobtrusive as possible, to have the program recover or fail gracefully.**
- **Some guidelines for deciding which defensive programming tools to leave in your production code and which to leave out:**
  - (1) Leave in code that checks for important errors
    - Decide which areas of the program can afford to have undetected errors and which areas cannot.
    - E.g., in a spreadsheet program, undetected errors in the screen-update area of the program may be affordable, but undetected errors in the calculation engine is unaffordable.

(2) Remove code that checks for trivial errors

- **If an error has truly trivial consequences, remove code that checks for it.**
- **In the previous example, you might remove the code that checks the spreadsheet screen update.**
- **"Remove" doesn't mean physically remove the code. It means use version control, precompiler switches, or some other technique to compile the program without that particular code.**
- **If space isn't a problem, you could leave in the error-checking code but have it log messages to an error-log file unobtrusively.**

(3) Remove code that results in hard crashes

- **If your program contains debugging code that could cause a loss of data, take it out of the production version.**
- **During development, when your program detects an error, you'd like the error to be as noticeable as possible so that you can fix it.**
  - Often, the best way to accomplish such a goal is to have the program print a debugging message and crash when it detects an error.
  - This is useful even for minor errors.
- **During production, your users need a chance to save their work before the program crashes and are probably willing to tolerate a few anomalies in exchange for keeping the program going long enough for them to do that.**
  - Users don't appreciate anything that results in the loss of their work, regardless of how much it helps debugging and ultimately improves the quality of the program.

(4) Leave in code that helps crash gracefully

- **If your program contains debugging code that detects potentially fatal errors, leave the code in that allows the program to crash gracefully.**
- **In the Mars Pathfinder, for example, engineers left some of the debug code in by design. An error occurred after the Pathfinder had landed.**
- **By using the debug aids that had been left in, engineers at JPL were able to diagnose the problem and upload revised code to the Pathfinder, and the Pathfinder completed its mission perfectly (March 1999).**

(5)(6) Log errors and friendly error messages

- **5. Log errors for your technical support personnel**
  - Consider leaving debugging aids in the production code but changing their behavior so that it's appropriate for the production version.
  - If you've loaded your code with assertions that halt the program during development, you might considering changing the assertion routine to log messages to a file during production rather than eliminating them altogether.

- **6. See that the error messages you leave in are friendly**
  - If you leave internal error messages in the program, verify that they're in language that's friendly to the user.
  - "You've got a bad pointer allocation!" ????
  - A common and effective approach is to notify the user of an "internal error" and list an email address or phone number the user can use to report it.

Being Defensive about Defensive Programming

- **Too much defensive programming creates problems of its own.**
  - Overly defensive programming however introduces unnecessary code for errors impossible to even happen, adds complexity to the software, thus wasting runtime and maintenance costs.
  - Code installed for defensive programming is not immune to defects, and you're just as likely to find a defect in defensive-programming code as in any other code.
  - There is also the risk that the code traps or prevents too many exceptions, potentially resulting in unnoticed, incorrect results.

- **Think about where you need to be defensive, and set your defensive-programming priorities accordingly.**

CHECKLIST: Defensive Programming

- **General**
  - Does the routine protect itself from bad input data?
  - Have you used assertions to document assumptions, including preconditions and postconditions?
  - Have assertions been used only to document conditions that should never occur?
  - Does the architecture or high-level design specify a specific set of error handling techniques?
  - Does the architecture or high-level design specify whether error handling should favor robustness or correctness?
  - Have barricades been created to contain the damaging effect of errors and reduce the amount of code that has to be concerned about error processing?

CHECKLIST: Defensive Programming

- **General**
  - Have debugging aids been used in the code?
  - Has information hiding been used to contain the effects of changes so that they won't affect code outside the routine or class that's changed?
  - Have debugging aids been installed in such a way that they can be activated or deactivated without a great deal of fuss?
  - Is the amount of defensive programming code appropriate—neither too much nor too little?
  - Have you used offensive programming techniques to make errors difficult to overlook during development?

CHECKLIST: Defensive Programming

- **Exceptions**
  - Has your project defined a standardized approach to exception handling?
  - Have you considered alternatives to using an exception?
  - Is the error handled locally rather than throwing a non-local exception if possible?
  - Does the code avoid throwing exceptions in constructors and destructors?
  - Are all exceptions at the appropriate levels of abstraction for the routines that throw them?
  - Does each exception include all relevant exception background information?
  - Is the code free of empty catch blocks? (Or if an empty catch block truly is appropriate, is it documented?)

CHECKLIST: Defensive Programming

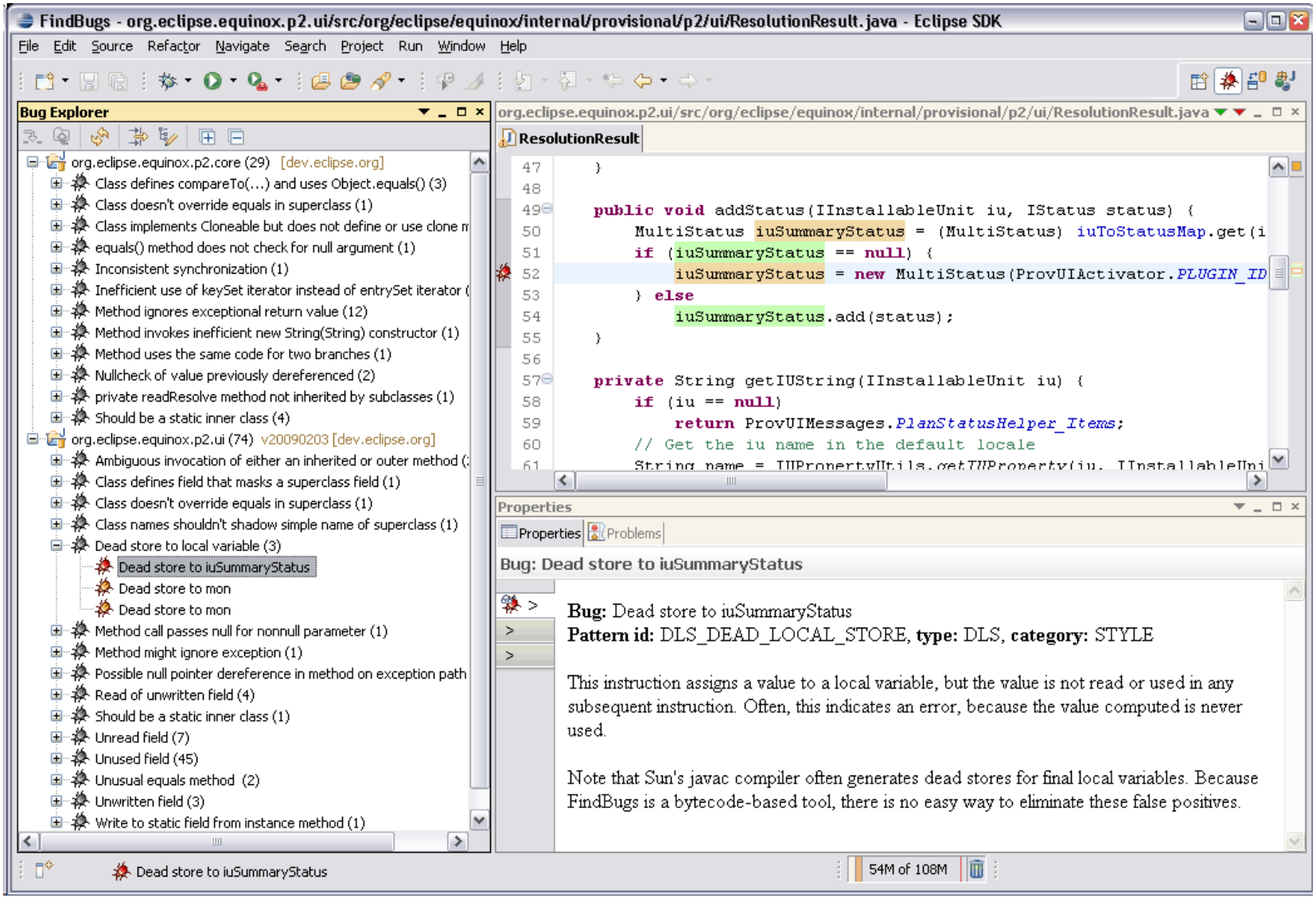
- **Security Issues**
  - Does the code that checks for bad input data check for attempted buffer overflows, SQL injection, html injection, integer overflows, and other malicious inputs?
  - Are all error-return codes checked?
  - Are all exceptions caught?
  - Do error messages avoid providing information that would help an attacker break into the system?

4 The FindBugs tool

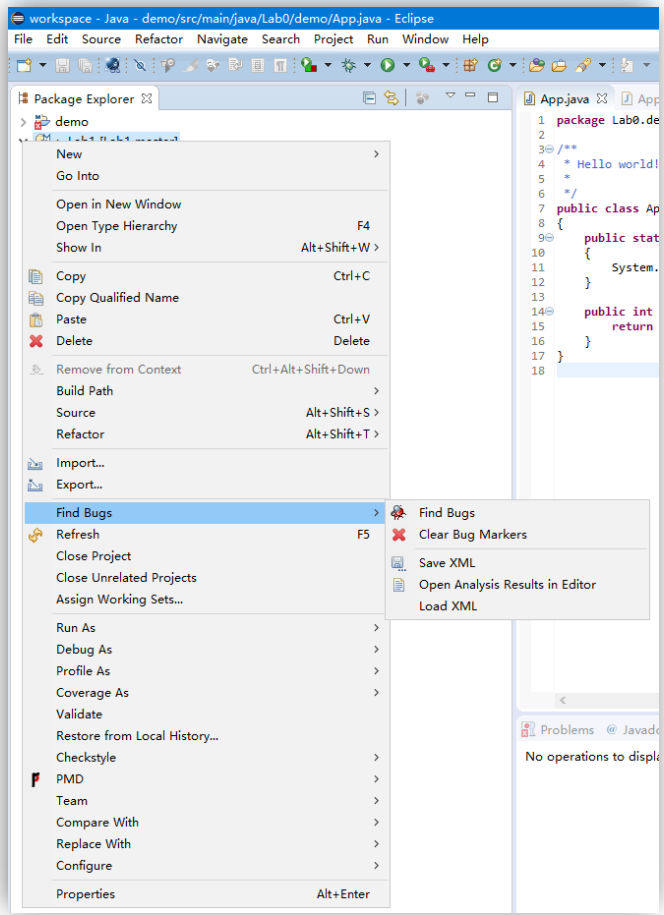
FindBugs

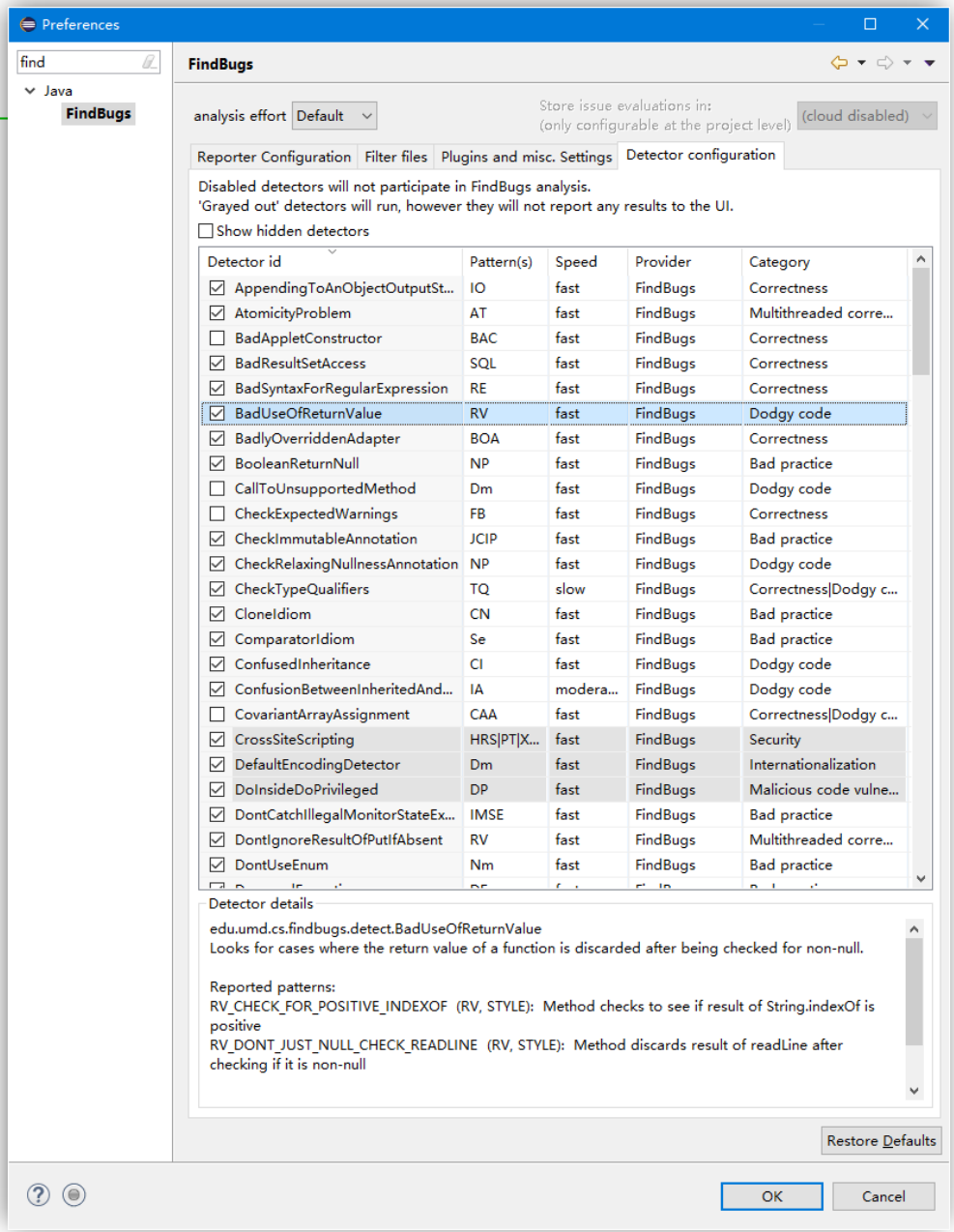
- **FindBugs** is a program which uses static analysis to look for bugs in Java code. Java静态代码分析工具
  - It operates on Java bytecode.
- Potential errors are classified in four ranks: scariest, scary, troubling and of concern. This is a hint to the developer about their possible impact or severity.
  - Bug list can be found in http://findbugs.sourceforge.net/bugDescriptions.html
- It is distributed as a stand-alone GUI application, but also plug-ins available for Eclipse, Gradle, Maven, and Jenkins.
  - http://findbugs.sourceforge.net/
  - http://findbugs.cs.umd.edu/eclipse/

FindBugs


FindBugs - org.eclipse.equinox.p2.ui/src/org/eclipse/equinox/internal/provisional/p2/ui/ResolutionResult.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Bug Explorer
org.eclipse.equinox.p2.core (29) [dev.eclipse.org]
Class defines compareTo(...) and uses Object.equals() (3)
Class doesn't override equals in superclass (1)
Class implements Cloneable but does not define or use clone m
equals() method does not check for null argument (1)
Inconsistent synchronization (1)
Inefficient use of keySet iterator instead of entrySet iterator (
Method ignores exceptional return value (12)
Method invokes inefficient new String(String) constructor (1)
Method uses the same code for two branches (1)
Nullcheck of value previously dereferenced (2)
private readResolve method not inherited by subclasses (1)
Should be a static inner class (4)
org.eclipse.equinox.p2.ui (74) v20090203 [dev.eclipse.org]
Ambiguous invocation of either an inherited or outer method (:
Class defines field that masks a superclass field (1)
Class doesn't override equals in superclass (1)
Class names shouldn't shadow simple name of superclass (1)
Dead store to local variable (3)
Dead store to iuSummaryStatus
Dead store to mon
Dead store to mon
Method call passes null for nonnull parameter (1)
Method might ignore exception (1)
Possible null pointer dereference in method on exception path
Read of unwritten field (4)
Should be a static inner class (1)
Unread field (7)
Unused field (45)
Unusual equals method (2)
Unwritten field (3)
Write to static field from instance method (1)
org.eclipse.equinox.p2.ui/src/org/eclipse/equinox/internal/provisional/p2/ui/ResolutionResult.java
ResolutionResult
47 }
48
49 public void addStatus(IInstallableUnit iu, IStatus status) {
50 MultiStatus iuSummaryStatus = (MultiStatus) iuToStatusMap.get(i
51 if (iuSummaryStatus == null) {
52 iuSummaryStatus = new MultiStatus(ProvUIActivator.PLUGIN_ID
53 } else
54 iuSummaryStatus.add(status);
55 }
56
57 private String getIUString(IInstallableUnit iu) {
58 if (iu == null)
59 return ProvUIMessages.PlanStatusHelper_Items;
60 // Get the iu name in the default locale
Properties
Properties Problems
Bug: Dead store to iuSummaryStatus
Bug: Dead store to iuSummaryStatus
Pattern id: DLS_DEAD_LOCAL_STORE, type: DLS, category: STYLE
This instruction assigns a value to a local variable, but the value is not read or used in any subsequent instruction. Often, this indicates an error, because the value computed is never used.
Note that Sun's javac compiler often generates dead stores for final local variables. Because FindBugs is a bytecode-based tool, there is no easy way to eliminate these false positives.
Dead store to iuSummaryStatus
54M of 108M

FindBugs


workspace - Java - demo/src/main/java/Lab0/demo/App.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
demo
App.java
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Find Bugs
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As
Profile As
Coverage As
Validate
Restore from Local History...
Checkstyle
PMD
Team
Compare With
Replace With
Configure
Properties Alt+Enter
Find Bugs
Clear Bug Markers
Save XML
Open Analysis Results in Editor
Load XML
Problems
No operations to displ


Preferences
find
Java
FindBugs
FindBugs
analysis effort Default
Store issue evaluations in:
(only configurable at the project level)
(cloud disabled)
Reporter Configuration
Filter files
Plugins and misc. Settings
Detector configuration
Disabled detectors will not participate in FindBugs analysis.
'Grayed out' detectors will run, however they will not report any results to the UI.
Show hidden detectors
Detector id
Pattern(s)
Speed
Provider
Category
AppendingToAnObjectOutputSt... IO fast FindBugs Correctness
AtomicityProblem AT fast FindBugs Multithreaded corre...
BadAppletConstructor BAC fast FindBugs Correctness
BadResultSetAccess SQL fast FindBugs Correctness
BadSyntaxForRegularExpression RE fast FindBugs Correctness
BadUseOfReturnValue RV fast FindBugs Dodgy code
BadlyOverriddenAdapter BOA fast FindBugs Correctness
BooleanReturnNull NP fast FindBugs Bad practice
CallToUnsupportedMethod Dm fast FindBugs Dodgy code
CheckExpectedWarnings FB fast FindBugs Correctness
CheckImmutableAnnotation JCIP fast FindBugs Bad practice
CheckRelaxingNullnessAnnotation NP fast FindBugs Dodgy code
CheckTypeQualifiers TQ slow FindBugs Correctness|Dodgy c...
CloneIdiom CN fast FindBugs Bad practice
ComparatorIdiom Se fast FindBugs Bad practice
ConfusedInheritance CI fast FindBugs Dodgy code
ConfusionBetweenInheritedAnd... IA modera... FindBugs Dodgy code
CovariantArrayAssignment CAA fast FindBugs Correctness|Dodgy c...
CrossSiteScripting HRS|PT|X... fast FindBugs Security
DefaultEncodingDetector Dm fast FindBugs Internationalization
DoInsideDoPrivileged DP fast FindBugs Malicious code vulne...
DontCatchIllegalMonitorStateEx... IMSE fast FindBugs Bad practice
DontIgnoreResultOfPutIfAbsent RV fast FindBugs Multithreaded corre...
DontUseEnum Nm fast FindBugs Bad practice
Detector details
edu.umd.cs.findbugs.detect.BadUseOfReturnValue
Looks for cases where the return value of a function is discarded after being checked for non-null.
Reported patterns:
RV_CHECK_FOR_POSITIVE_INDEXOF (RV, STYLE): Method checks to see if result of String.indexOf is positive
RV_DONT_JUST_NULL_CHECK_READLINE (RV, STYLE): Method discards result of readLine after checking if it is non-null
Restore Defaults
OK
Cancel

Summary

Summary

- **Recall: Designing an ADT**
- **Assertions**
  - What to Assert and What not to?
  - Guidelines for Using Assertions
- **Defensive Programming**
  - Techniques for defensive programming
- **The FindBugs tool**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

May 6, 2018