Assessing Java Coding Vulnerabilities in Undergraduate Software Engineering Education by Using Open Source Vulnerability Analysis Tools

Yen-Hung Hu
yhu@nsu.edu

Thomas Kofi Annan
t.k.annan@spartans.nsu.edu

Department of Computer Science
Norfolk State University
Norfolk, Virginia

Abstract - Security and quality are two vital attributes of any software application no matter how infinitesimal it might be. Tackling a software problem by its source is one of the most trusted models used in problem solving approaches. In this paper, we want to ensure that all undergraduate Java learners write codes based on the security and quality guidelines expected in the industry right from the day they start learning "HelloWorld!" in Java. In the research, sample codes getting from several Java books used in teaching Java concepts for undergraduate courses were used as the case study. These sample codes were tested using an open source tool developed based on security and quality guidelines. The tool determines the vulnerability level in any Java code passed as an input to it then it analyzes the code and generates a report indicating the threat level based on the vulnerabilities in the code. The results of this paper will be published and authors of the selected books for the research will be notified with those vulnerabilities in their source codes along with suggestions for fixing those vulnerabilities.

Categories and Subject Descriptors

K.6.5 [Management of Computing and Information Systems]: *Security and Protection*

General Terms

Software and Vulnerability

Keywords

Security, Quality, Java, Vulnerability, Open Source

1. INTRODUCTION

The advances in technology and Internet connectivity, combined with ever increasing number of threats and attacks, require software security and quality to be integrated into the traditional verification and validation process. The report [1] disclosed that there were 1,208 vulnerabilities discovered in 27 products of the top 50 portfolio and 76% of the vulnerabilities in the 50 most popular programs on private PCs in 2013 were from third-party programs. In 2014, the NIST added about 7900 new vulnerabilities into the National Vulnerability Database (NVD). Of these, 24% were labeled high severity [2]. Fixing vulnerabilities before shipment can no more be considered optional? Most of the reported software vulnerabilities are leftovers forgotten by developers, thought to be benign codes. Such kind of mistakes can survive unaudited for years until they end up exploited by hackers [3].

Software vulnerability is a threat to confidentiality, integrity and availability of the information used or generated by software. This threat can only be limited if coding standards are embedded as the core in every software development course in undergraduate education. To deliver secure and quality software, vulnerabilities must be identified in source code. With the phenomenal growth of the Internet, it is imperative to test for the security and quality of software during its developmental lifecycle and fix the vulnerabilities, if any is found, before deployment [4]. Software security and quality deal with shielding software from hits by hackers, and ensures that the software continues to work in spite of threats [5]. Many organizations develop, run, and maintain numerous systems for which one or more security and quality attributes (i.e., confidentiality, integrity, availability, authentication, authorization, and non-repudiation) are of vital importance.

Software vulnerabilities are often the result of bad programming practices during development. Some of these vulnerabilities are easily detected and fixed when the program crashes or unexpected output is given. Others will never be noticed during normal use. Automatic source code analyzers can help detect these vulnerabilities before deployment of a software system. The design and implementation of such a tool was the original idea of this paper but we ended up adopting an open source version of what we wanted to develop. In choosing this tool, we considered software security and quality assurance methodology, focusing on identifying and correcting potential problems during the software development process as opposed to patching broken systems in the production environment. Since the main focus is churning out security and quality conscious Java programmers in undergraduate programs, the tool will be used in testing most of the codes from the popular Java books, known in the educational domain.

In this paper, we focus on testing for software vulnerabilities using source code analysis (also invariably referred to as static code analysis). Static code analysis refers to examining a piece of code without actually executing it. The technique of evaluating software during its execution is referred to as run-time code analysis (also called dynamic code analysis) – the other commonly used approach to test for software vulnerabilities [4]. While dynamic code analysis is popularly used to test for logical errors and stress test the software by running it in an environment with limited resources, static or source code analysis has been the principal means to evaluate the software with respect to functional, semantic and structural issues including, but not limited to, type checking, style checking, program verification, property checking and bug finding.

The remainder of this paper is organized as follows: Section 2 brings up our motivation. Section 3 defines software security and quality standards in Java. Section 4 explains our Java source code analysis and discusses the research results. Section 5 concludes this paper and points out future work. Section 6 expresses our recommendations.

2. MOTIVATION

Software security and quality deal with developing software with minimal to zero flaws. Security and quality are essential to give verification, reliability, accessibility and privacy. Software vulnerability can be seen as a flaw, a weakness, or even an error in the system that can be exploited by an attacker in order to alter the normal activity of the system. Because the number of software systems increases everyday also the number of vulnerabilities. Additionally, if we consider that most of the systems are exposed to multiple users (Internet) and environments (operating systems for example) then it is just a matter of time that someone can launch an attack (sequence of actions) whose consequences are unpredictable in damages and cost [6].

Until recently, security and quality have been often considered as an afterthought, and the bugs are mostly detected post-deployment through user experiences and attacks reported. The bugs are controlled through patch code (more formally called "updates") that is quite often sent to customers via the Internet. It is sometimes annoying to notice the icons with tagline "Security Alerts", "Updates are ready to be installed on your computer", when someone has just gone through a humongous waiting time for the most recent updates to be downloaded and installed.

Software applications have become part of our everyday life, from embedded devices to complex systems that control real time systems to nuclear reactors. Security and quality in these applications and services cannot be overruled. Most Software applications even though the original idea is to solve one particular problem but will end up having more features as the application matures through its growth life cycle. Every new added feature of any software comes with a potential risk to the overall application lifecycle. In view of this change, security and quality standards need to be established during the development cycle. Most software's are vulnerable not because the companies don't follow security and quality standards but rather the developers are not exposed to these standards early enough during application development training stages. These stages can be self-taught or a formal training from College if one happens to be a computer science major or any discipline with programming module embedded. Undergraduates in these domains need to be introduced to coding standards before and during the entire training. In this paper, we will look at three main things:

1. Knowing the basics of software security and quality standards in Java,
2. Designing software with security and quality features and
3. Developing software based on security and quality standards.

3. SOFTWARE SECURITY AND QUALITY STANDARDS IN JAVA

Java [7] was chosen for this study because: it's known to be secure compared to other programming languages because of its whole architecture and implementation of the Java virtual machine. Byte code verification is the backbone of the secure nature of Java. Simplicity of Java is seen in its small language and smaller interpreter but has rich set of libraries (i.e., API). It is fully object oriented by supporting encapsulation, inheritance, abstraction and polymorphism. Libraries for networking and remote method invocation make it more of a distributed system language, which support enterprise applications in the industry. Portability, architectural neutrality, multithreaded and dynamic nature of Java made it a language of choice for our study.

3.1 Coding Quality Standard

Secure coding standards cannot be achieved if we ignore quality coding standard. Main purpose of coding standards is to make code readable, understandable and clean to all programmers including the original author of the code. Some of the keys to code readability are [8]:

- Modularity: Code is broken into sections that are completely independent of each other, so that you can understand a block of code without looking at any other code.

- Cohesion: All code for a given task is in one block, with no other code intervening.
- Consistency: Code is written and formatted in a consistent manner, so that it is easy to browse. You can see clear patterns in the structure and indentation, so that you can quickly scan the code and find what you're looking for.

Observing an implementation of these quality coding standards does not only give you the benefits stated above but also prepares one to follow security conventions in coding.

3.2 Java Security Guidelines

In here, we only introduce ten Java security guidelines because they are particularly important for programmer and were selected in [9] [10] [11]. These security guidelines when flouted will be at risk of breaching confidentiality, integrity and availability that forms the core tenets of security principles in any domain. Key factors of the guidelines are described below.

- Be aware of numeric promotion behavior: Numeric promotions are used to convert mixed operands of an arithmetic operator to a common type such that the operation can be performed. When using arithmetic operators with mixed operand sizes, narrower operands will be promoted to the type of wider operands.
- Use the Same data type for the second and third operands in conditional expressions: In general, the form of a Java conditional expression can be represented as *operand_1 expression ? operand_2 expression: operand_3*. It uses the Boolean value of the first operand expression (operand_1) to determine which of the other two operand expressions (operand_2, operand_3) will be evaluated. If the first operand expression (operand_1) is true, then the second operand expression (operand_2) is chosen. However, the third operand expression (operand_3) will be chosen if the first operand expression (operand_1) is false. Since the result of a conditional expression is not constant due to its complicated rules, operands in the

second operand expression and those in the third operand expression should have identical data types.

- Avoid inadvertent wrapping of loop counters: A while or for loop may execute infinitely if the counter cannot reach its final value which is defined inappropriately, or the predefined conditional expression cannot be fulfilled.
- Strive for logical completeness: Logical incompleteness occurs when programmers fail to consider all possible data states. This type of error may lead the program to unexpected results if data states are not considered in the predefined conditional expressions.
- Do not confuse abstract object equality with reference equality: Java uses equality operators (i.e., == and !=) for testing reference equality but using equals() method for testing object equality. Some programmers may confuse the purpose of the == operation with that of the equals() method. In general, the == operator checks whether two references refer to the same object; in other words these two references refer to the same memory address. The equals() method identifies whether two objects have the same contents but may not be in the same memory address.
- Understand how escape characters are interpreted when strings are loaded: Java allows escape sequences in character and string literals for certain output format purposes. Correct use of escape sequences in string literals requires understanding how the escape sequences are interpreted by the Java compiler. Sometimes, there will be any subsequent processor such as SQL engine involved in the Java code. In order to pass escape sequence characters to the SQL engine correctly, an extra backslash (\) needs to be added in front of the escape sequence characters. The implementations involving with subsequent processors are complicated and should follow the instructions associated with those processors carefully.
- Use a try-with-resources statement to safely handle closeable resources: Java standard contains the try-with-resources code for safely handling

closeable resources. However, it must be implemented correctly to prevent problems such as failing to close a resources because an exception is thrown as a result of closing another resource or masking an important exception when a resource is closed.

- Do not expose methods that use reduced-security checks to untrusted code: In this guideline, any code that invokes methods using a reduced-security check must guarantee that these methods will not be invoked as a representative of untrusted codes. A reduced-security check method checks only the calling method which is authorized rather than checking every method in the call stack [9] [10] [11].
- Do not use the clone() method to copy untrusted method parameters: To enhance Java security, clone() has been used to copy mutable method parameters to mitigate potential security vulnerabilities. However, inappropriate use of the clone() method can leave security vulnerabilities that return unexpected results directed by attackers. For instance, attackers can provide arguments that appear normal but eventually are objects of malicious classes which extend from normal classes and override clone() method with malicious codes to bypass or lessen input validation and security checks. Several details on this issue have been studied in [11].
- Document thread-safety and use annotations where applicable: Two set of Java language annotations including Java Concurrency in Practice (JCIP) [12] and SureLogic [13] are available and useful for documenting design intent with respect to thread-safety. For instance, JCIP provides three-class level annotations for thread-safety. The @ThreadSafe denotes that no sequences of accesses can leave the object in inconsistence state regardless the interleaving of these accesses by the runtime or any external synchronization or coordination on the part of the caller. The @Region and @RegionLock annotations represent the locking policy upon which the promise of thread-safety is predicated. [11].

4. JAVA SOURCE CODE ANALYSIS

Initial idea of this study was to develop a Java vulnerability analysis tool (see Figure 1) to find bugs based on the Java security and quality guidelines.

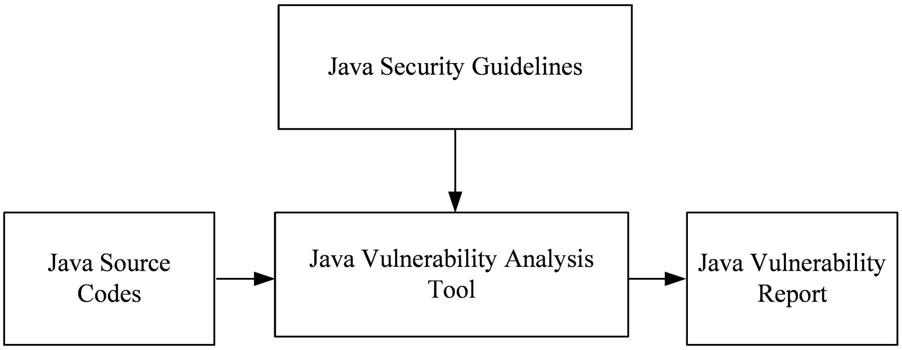

Figure 1: Java Vulnerability Analysis Tool

The analysis tool will include four modules: Java Source/Byte Codes Feeder, Java Security/Quality Guidelines Feeder, Java Vulnerability Analyzer and Java Vulnerability Reporter. They are described below:

- Java Source/Byte Codes Feeder: This is the module that is able to attach multiple Java source/byte codes and feed them into the Java Vulnerability Analyzer. This module will conduct preliminary investigations to eliminate inappropriate Java source/byte codes if existed.
- Java Vulnerability Analyzer: This module will analyze inputs from the Java Source/Byte Codes Feeder and rules from the Java Security/Quality Guidelines Feeder and generate outputs to the Java Vulnerability Reporter. This module will be implemented with multi-thread technology to improve the performance while multiple Java source/byte codes are carried from the Java Source/Byte Codes Feeder. The outputs of this module will include: potential vulnerabilities of each source/byte code

against the selected Java security/quality guidelines and suggested solutions for these vulnerabilities.

- Java Security/Quality Guidelines Feeder: This module will be built based on the adopted Java security and quality guidelines. The guidelines may be varied because different implementations require different security and quality measurements.
- Java Vulnerability Reporter: This module will summarize vulnerabilities of each source/byte code and demonstrate potential solutions for them. Meanwhile, it will accept the requests of generating different output file formats.

4.1 FindBugs

After reading around it was discovered that there is an open source tool FindBugs [15] that does almost what we needed to be implemented in our solution. This tool detects potential errors in Java programs. Possible bugs are classified in four ranks: (i) scariest, (ii) scary, (iii) troubling and (iv) of concern. This is a hint to the developer about their possible impact of severity and quality [16].

FindBugs operates on Java byte code, rather than source code. The software is distributed as a stand-alone GUI application. However, it can be also embedded into several popular platforms [15]. All the sample codes were compiled under one Java application name "BookNumber" in NetBeans integrated development environment (IDE) [17] and an executable was generated as BookNumber.jar which is Java archive. A new project was then created from the FindBugs from its user interface and the jar file was passed to it through a file chooser. The SampleCodes.jar was then analyzed with this tool.

4.2 Our Security and Quality Vulnerability Measurements vs Bugs FindBugs

FindBugs categorizes bugs into to 9 different categories: bad practice, correctness, experimental, internationalization, malicious code vulnerability, multithreaded correctness, performance, security, and dodgy code. In our measurement, 5 categories among them (i.e., bad practice, correctness, malicious code vulnerability, multithreaded corrected, and security) will apparently affect software security and all of them will affect software quality (see Table 1).

Our Measurements		
Security	**Quality**	**Bug in Findbugs in 9 categories**
X	X	Bad Practice
X	X	Correctness
	X	Experimental
	X	Internationalization
X	X	Malicious Code Vulnerability
X	X	Multithreaded Correctness
	X	Performance
X	X	Security
	X	Dodgy Code

Table 1: The relationship between our security and quality vulnerability measurement and categories of bugs in FindBugs

In the early stage of this research, instead of implementing our proposed tool, we adopted FindBugs to check potential bugs in the sample codes. Although its security and quality guidelines are not exactly equivalent to our proposed ones, FindBugs can still provide us very much information that benefits to our next research step: building our proposed Java vulnerability analysis tool.

4.3 Sample Codes

In this research, sample codes from popular Java books adopted during a normal undergraduate Java class were used. They covered most of the entire concepts in Java standard edition topics. Some of the codes are listed on [14].

4.4 Results from Analysis

In this research, we have fed the sample source codes from four different well-known books in the development community and academia into the FindBugs and examined their results (see Table 2).

Overall the FindBugs detected 345 bugs in the sample codes, out of these 22 were due to correctness, 50 bad practices, 39 internationalization, 82 performances, 2 securities and 149 Dodgy codes. Apparently, 74 bugs will affect software security, and all of these 345 bugs will affect software quality (see Table 2).

Bugs in FindBugs	**Book 1**	**Book 2**	**Book 3**	**Book 4**
Correctness	3	5	11	3
Bad practices	14	8	5	23
Security	0	0	1	1
Internationalization	13	6	11	9
Performance	13	16	41	12
Dodgy code	39	1	51	58
Total bugs	82	36	121	106

Table 2: Summary of bugs in each book

We then rewrote Table 2 to reflect our security and quality vulnerability measurements (see Table 3).

Our Measurement	Book 1	Book 2	Book 3	Book 4
Security Vulnerability	17	13	17	26
Quality Vulnerability	82	36	121	106

Table 3: Our security and quality vulnerability measurements of selected books

To help future developers, we specified bug pattern, description of the pattern and recommendation for solving the bug in the first column and located sources of the bug in the second column of Table 4 (in Appendix).

These results prompt us that instructors and authors must incorporate these securities and best practices standards in their lessons and code samples at all times to instill these principles to learners of programming languages. Also, online website tutorials and forums needs to be advocate for these as well by checking against these codes.

5. CONCLUSIONS

In this paper, we studied Java security and quality guidelines, proposed a Java vulnerability analysis tool and investigated vulnerabilities in the sample source codes. In the early stage of this research, we adopted FindBugs to examine the sample source codes. We identified their patterns, described the reasons, and addressed our recommendations. The research results indicated there are many common bugs in the sample source codes, which also raise the security concerns, listed on the Java security guidelines and will need further studies. In the next step of this research, we plan to build our own analysis tool, involve more sample source codes and design our own security and quality enhanced curricula for undergraduate software engineering education. Online programming tutorials site and forums needs to be advocates for such secure and quality coding practices as well.

6. RECOMMENDATIONS

Learners of any programming language are the future of that language in which Java is not an exception, in view of that instructor from every institution should try as much as possible to instill and be advocate secure code practices for their students by ensuring that the materials they use have embedded these secure and quality coding practices in all the source of information and also their sample codes as well. Furthermore, the instructors must ensure that student are very conscious about them in their code submission for all coding assignments and quizzes.

Authors of programming books are the main source of information for beginners and professionals of any programming language, they should play a part by adopting security and quality first before concept in writings and dissemination of knowledge in the books and documents. All authors can form a coalition to ensure other members of the group follow those security and quality guidelines needed to be shared with the learners at all times.

Online resources and code sharing websites like stack overflow needs to be proponent of these security and quality guidelines by having a third-party code pasting editor with security and quality analysis capability, which will share vulnerabilities in any code pasted with that editor. In the future version of this research will propose that such a tool will be developed and shared among the coding community as well as educational institutions.

REFERENCES

[1] Secunia Vulnerability Review 2014, February 2014, https://secunia.com/?action=fetch&filename=secunia_vulnerability_review_2014.pdf .

[2] Dorothy E. Denning, "Privacy and security: Toward More Secure Software", Communications of the ACM, Vol. 58 No. 4, Pages 24-26, April 2015.

[3] Marco Guarnieri, Paul El Khoury and Gabriel Serme "Security vulnerabilities detection and protection using Eclipse", in Proceeding of ECLIPSE-IT 2011, 6th Workshop of the Italian Eclipse Community, September 22-23, 2011, Milano, Italy.

[4] Natarajan Meghanathan, "Identification and Removal of Software Security Vulnerabilities using Source Code Analysis: A Case Study on a Java File Writer Program with Password Validation Features", International Journal of Network Security & Its Applications (IJNSA), Vol.5, No.1, January 2013.

[5] K. P. Lavanya1, B. Vishwanatha1 and Anirban Basu1, "Detection and Correction of Software Vulnerabilities in Java Code", International Journal of Current Research in Science and Technology, Volume 1, Issue 7, Pages 19-27, 2015.

[6] Willy Jimenez, Amel Mammar and Ana Cavalli, "Software Vulnerabilities, Prevention and Detection Methods: A Review", in Proceeding of SEC-MDA workshop, 24-June 2009, Enschede, The Netherlands.

[7] Oracle, "Introduction to Java Platform, Enterprise Edition 7", Oracle White Paper, June 2013. http://www.oracle.com/technetwork/java/javaee/javaee7-whitepaper-1956203.pdf

[8] Lecture note of CS315 Computer Organization and Assembly Language Programming, Department of Computer Science, University of Wisconsin - Milwaukee, http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch07.html

[9] SEI, CMU, "Top 10 Coding Guidelines for Java", http://www.sei.cmu.edu/news/article.cfm?assetid=77817.

[10] Fred Long, Dhruv Mohindra, Robert Seacord, Dean F. Sutherland, David Svoboda, Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs, Addison-Wesley Professional, Aug. 30, 2013.

[11] SEI, CMU, "SEI CERT Oracle Coding Standard for Java", https://www.securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

[12] Goetz, Brian, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency in Practice. Boston: Addison-Wesley Professional, 2006.

[13] SureLogic, http://surelogic.com/

[14] Link to sample codes https://github.com/thomaskofiannan/SampleCodes

[15] FindBugs - Find Bugs in Java Programs, http://findbugs.sourceforge.net/

[16] https://en.wikipedia.org/wiki/FindBugs

[17] NetBeans IDE, https://netbeans.org

APPENDIX

Bugs	**Source in Code**	**Book 1**

Pattern: Array index is out of bounds
 	
Recommendation: Array operation is performed, but array index is out of bounds, which will result in ArrayIndexOutOfBoundsException at runtime.
	Try {
	    if (a == 1) a = a / (a - a);
	    if (a == 2) {
	        int c[] = {1};
	        c[42] = 99;
	}	

Pattern: Comparison of String objects using == or != Recommendation: This code compares java.lang.String objects for reference equality using the == or != operators. Unless both strings are either constants in a source file, or have been interned using the String.intern() method, the same string value may be represented by two different String objects. Consider using the equals(Object) method instead.	System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));	

Pattern: Rough value of known constant found
 	
Recommendation: It's recommended to use the predefined library constant for code clarity and better precision.
	pi = 3.1416;
	a = pi * r * r;	

Bugs	Source in Code	Book 1

Pattern: Consider using Locale parameterized version of invoked method
 	
Recommendation: A String is being converted to upper or lowercase, using the platform's default encoding. This may result in improper conversions when used with international characters. Use these two instead.
 	
String.toUpperCase (Locale l)
 	
String.toLowerCase (Locale l)
	String upper = s.toUpperCase();
	
	String lower = s.toLowerCase();	

Pattern: Reliance on default encoding Recommendation: Found a call to a method which will perform a byte to String (or String to byte) conversion, and will assume that the default platform encoding is suitable. This will cause the application behaviour to vary between platforms. Use an alternative API and specify a charset name or Charset object explicitly.	byte buf[] = str.getBytes();	

Bugs	**Source in Code**	**Book 1**
Pattern: Use the nextInt method of Random rather than nextDouble to generate a random integer Recommendation: If r is a java.util.Random, you can generate a random number from 0 to n-1 using r.nextInt(n), rather than using (int)(r.nextDouble() * n). The argument to nextInt must be positive. If, for example, you want to generate a random value from -99 to 0, use -r.nextInt(100).	int prob = (int) (100 * rand.nextDouble());	
Pattern: Method invokes inefficient new String(String) constructor Recommendation: Using the java.lang.String(String) constructor wastes memory because the object so constructed will be functionally indistinguishable from the String passed as a parameter. Just use the argument String directly.	String s2 = new String(s1);	

Pattern: Inefficient use of keySet iterator instead of entrySet iterator
 	
Recommendation: This method accesses the value of a Map entry, using a key that was retrieved from a keySet iterator. It is more efficient to use an iterator on the entrySet of the map, to avoid the Map.get(key) lookup.
	Iterator<String> itr = set.iterator();
	
	while(itr.hasNext()) {
	
	    str = itr.next();
	
	System.out.println(str + ": " +
	
	    balance.get(str));
	
	}	

Bugs	**Source in Code**	**Book 1**

Pattern: Method invokes inefficient floating-point Number constructor; use static valueOf instead
 	
Recommendation: Using new Double(double) is guaranteed to always result in a new object whereas Double.valueOf(double) allows caching of values to be done by the compiler, class library, or JVM. Using of cached values avoids object allocation and the code will be faster. Unless the class must be compatible with JVMs predating Java 1.5, use either autoboxing or the valueOf() method when creating instances of Double and Float.
	TreeMap<String, Double> tm = new
	    TreeMap<String, Double>();
	tm.put("John Doe", new Double(3434.34));	

Pattern: Unsigned right shift cast to short/byte Recommendation: The code performs an unsigned right shift, whose result is then cast to a short or byte, which discards the upper bits of the result. Since the upper bits are discarded, there may be no difference between a signed and unsigned right shift (depending upon the size of the shift).	byte d = (byte) (b >>> 4);	

Bugs	**Source in Code**	**Book 1**

Pattern: Integral division result cast to double or float
 	
Recommendation: This code casts the result of an integral division (e.g., int or long division) operation to double or float. Doing division on integers truncates the result to the integer value closest to zero. The fact that the result was cast to double suggests that this precision should have been retained. What was probably meant was to cast one or both of the operands to double before performing the division. Here is an example:
 	
`int x = 2;`
`int y = 5;`
`// Wrong: yields result 0.0`
`double value1 =  x / y;`
`// Right: yields result 0.4`
`double value2 =  x / (double) y;`
	`byte b = 42;`
	`char c = 'a';`
	`short s = 1024;`
	`int i = 50000;`
	`float f = 5.67f;`
	`double d = .1234;`
	`double result = (f * b) + (i / c) - (d * s);`	

Bugs	**Source in Code**	**Book 1**

Pattern: Dead store to local variable
 	
Recommendation: 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 byte code-based tool, there is no easy way to eliminate these false positives.
	public class CallingCons {
	    public static void main(String args[]) {
	        C c = new C();
	        Protection2 ob1 = new Protection2();
	        OtherPackage ob2 = new
	           OtherPackage();
	    }
	}	

Pattern: Code contains a hard coded reference to an absolute pathname Recommendation: This code constructs a File object using a hard coded to an absolute pathname (e.g., new File("/home/dannyc/workspace/j2ee/src/share/com/sun/enterprise/deployment");	try (FileInputStream f = new FileInputStream("C:\\Users\\User\\Desktop\\FileInputStreamDemo.java")) {}	

Bugs	**Source in Code**	**Book 1**

Pattern: Dereference of the result of readLine() without nullcheck
 	
Recommendation: The result of invoking readLine() is dereferenced without checking to see if the result is null. If there are no more lines of text to read, readLine() will return null and dereferencing that will generate a null pointer exception.
	number = br.readLine();
	
	ht.put(name, number);	

Bugs	**Source in Code**	**Book 1**

Pattern: Switch statement found where default case is missing. This method contains a switch statement where default case is missing. Usually you need to provide a default case.
 	
Recommendation: Because the analysis only looks at the generated byte code, this warning can be incorrect triggered if the default case is at the end of the switch statement and the switch statement doesn't contain break statements for other cases.
	switch (result) {
	    case NO:
	        System.out.println("No");
	        break;
	    case YES:
	        System.out.println("Yes");
	        break;
	    case MAYBE:
	        System.out.println("Maybe");
	        break;
	    case LATER:
	        System.out.println("Later");
	        break;
	    case SOON:
	        System.out.println("Soon");
	        break;
	    case NEVER:
	        System.out.println("Never");
	        break;
	}	

Bugs	Source in Code	Book 2
Pattern: Bad attempt to compute absolute value of signed random integer Recommendation: This code generates a random signed integer and then computes the absolute value of that random integer. If the number returned by the random number generator is Integer.MIN_VALUE, then the result will be negative as well (since Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE). (Same problem raised for long values as well).	radius = Math.abs(generator.nextInt())%50 + 25;	
Pattern: Method call passes null for non-null parameter Recommendation: This method call passes a null value for a non-null method parameter. Either the parameter is annotated as a parameter that should always be non-null, or analysis has shown that it will always be dereferenced.	music[6] = JApplet.newAudioClip(url6);	
Pattern: Method might ignore exception Recommendation: This method might ignore an exception. In general, exceptions should be handled or reported in some way, or they should be thrown out of the method.	catch (Exception exception) {}	

Bugs	**Source in Code**	**Book 2**

Pattern: Certain swing methods needs to be invoked in Swing thread
 	
Recommendation: (From JDC Tech Tip): The Swing methods show(), setVisible(), and pack() will create the associated peer for the frame. With the creation of the peer, the system creates the event dispatch thread. This makes things problematic because the event dispatch thread could be notifying listeners while pack and validate are still processing. This situation could result in two threads going through the Swing component-based GUI -- it's a serious flaw that could result in deadlocks or other related threading issues. A pack call causes components to be realized. As they are being realized (that is, not necessarily visible), they could trigger listener notification on the event dispatch thread.
	JFrame frame = new JFrame("Java Juke Box");
	
	frame.pack();	

Bugs	**Source in Code**	**Book 2**

Pattern: Method concatenates strings using + in a loop
 	
Recommendation: The method seems to be building a String using concatenation in a loop. In each iteration, the String is converted to a StringBuffer/StringBuilder, appended to, and converted back to a String. This can lead to a cost quadratic in the number of iterations, as the growing string is recopied in each iteration.
 	
Better performance can be obtained by using a StringBuffer (or StringBuilder in Java 1.5) explicitly.
 	
For example:
 	
// This is bad
 	
String s = "";
 	
for (int i = 0; i < field.length; ++i) {
 	
s = s + field[i];
 	
}
	String contents = "\nShopping Cart\n";
	
	contents += "\nItem\t\tUnit
	
	Price\tQuantity\tTotal\n";
	
	for (int i = 0; i < itemCount; i++)
	
	contents += cart[i].toString() + "\n";	

Bugs	Source in Code	Book 3

Pattern: instanceof will always return false
 	
Recommendation: This instanceof test will always return false. Although this is safe, make sure it isn't an indication of some misunderstanding or some other logic error.
	Employee employee1 = new Employee();
	
	SalaryEmployee employee2 = new SalaryEmployee();
	
	System.out.println("Employee1 is a
	
	SalaryEmployee: " + (employee1 instanceof
	
	SalaryEmployee));	

Pattern: A parameter is dead upon entry to a method but overwritten
 	
Recommendation: The initial value of this parameter is ignored, and the parameter is overwritten here. This often indicates a mistaken belief that the write to the parameter will be conveyed back to the caller.
	Length of arr2: " + arr2.length);
	
	private static void changeArray(int arr[]) {
	
	arr = new int[100];
	
	System.out.println("Length of arr: " +
	
	arr.length);
	
	}	

Pattern: Method may fail to close stream Recommendation: The method creates an IO stream object, does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. It is generally a good idea to use a finally block to ensure that streams are closed.	FileReader fileReader = new FileReader(file);	

Bugs	**Source in Code**	**Book 3**
Pattern: Boxing/unboxing to parse a primitive Recommendation: A boxed primitive is created from a String, just to extract the unboxed primitive value. It is more efficient to just call the static parseXXX method.	int num1 = Integer.valueOf("540").intValue();	

Pattern: Hardcoded constant database password
 	
Recommendation: This code creates a database connect using a hardcoded, constant password. Anyone with access to either the source code or the compiled code can easily learn the password.
	try (Connection connection =
	    DriverManager.getConnection(
	        "jdbc:mysql://localhost:3306/",
	        "root", "explore");	

Pattern: Call to equals(null)
 	
Recommendation: This method calls equals(Object), passing a null value as the argument. According to the contract of the equals() method, this call should always return false.
	Item item4 = null;
	System.out.println("item1 equals item4: " +
	    item1.equals(item4));	

Pattern: Hardcoded constant database password Recommendation: This code creates a database connect using a hardcoded, constant password. Anyone with access to either the source code or the compiled code can easily learn the password.	Connection con = DriverManager.getConnection("jdbc:derby://localhost:1527/contact", "userName", "password");	

Bugs	**Source in Code**	**Book 3**

Pattern: Method may fail to close database resource
 	
Recommendation: The method creates a database resource (such as a database connection or row set), does not assign it to any fields, pass it to other methods, or return it, and does not appear to close the object on all paths out of the method. Failure to close database resources on all paths out of a method may result in poor performance, and could cause the application to have problems communicating with the database.
	try {
	Connection con =
	DriverManager.getConnection(
	"jdbc:derby://localhost:1527/contact",
	"userName", "password");
	System.out.println("Schema: " +
	con.getSchema());
	DatabaseMetaData metaData =
	con.getMetaData();
	System.out.println("Auto Generated
	Keys: " +
	metaData.generatedKeyAlwaysReturned());
	} catch (SQLException ex) {
	ex.printStackTrace();
	}	

Bugs	**Source in Code**	**Book 3**

Pattern: Class implements same interface as superclass
 	
Recommendation: This class declares that it implements an interface that is also implemented by a superclass. This is redundant because once a superclass implements an interface, all subclasses by default also implement this interface. It may point out that the inheritance hierarchy has changed since this class was created, and consideration should be given to the ownership of the interface's implementation.
	private static class Player extends Entity implements Runnable {
	    @Override
	    public String toString() {
	        return "Player #" + id;
	    }
	}	

Bugs	**Source in Code**	**Book 4**

Pattern: Call to equals(null)
 	
Recommendation: This method calls equals(Object), passing a null value as the argument. According to the contract of the equals() method, this call should always return false.
	Item item4 = null;
	System.out.println("item1 equals item4: " +
	    item1.equals(item4));	

Bugs	**Source in Code**	**Book 4**
Pattern: Hardcoded constant database password Recommendation: This code creates a database connect using a hardcoded, constant password. Anyone with access to either the source code or the compiled code can easily learn the password.	Connection con = DriverManager.getConnection("jdbc:derby://localhost:1527/contact", "userName", "password");	
Pattern: Method may fail to close database resource Recommendation: The method creates a database resource (such as a database connection or row set), does not assign it to any fields, pass it to other methods, or return it, and does not appear to close the object on all paths out of the method. Failure to close database resources on all paths out of a method may result in poor performance, and could cause the application to have problems communicating with the database.	try { Connection con = DriverManager.getConnection("jdbc:derby://localhost:1527/contact", "userName", "password"); System.out.println("Schema: " + con.getSchema()); DatabaseMetaData metaData = con.getMetaData(); System.out.println("Auto Generated Keys: " + metaData.generatedKeyAlwaysReturned()); } catch (SQLException ex) { ex.printStackTrace(); }	

Bugs	**Source in Code**	**Book 4**

Pattern: Class implements same interface as superclass
 	
Recommendation: This class declares that it implements an interface that is also implemented by a superclass. This is redundant because once a superclass implements an interface, all subclasses by default also implement this interface. It may point out that the inheritance hierarchy has changed since this class was created, and consideration should be given to the ownership of the interface's implementation.
	private static class Player extends Entity implements Runnable {
	@Override
	public String toString() {
	return "Player #" + id;
	}
	}	

Table 4: Bug patterns and source codes of those bugs in each book and recommendations for fixing them