Code Smells, Refactoring and Coding Standards

Week 11 and Week 12

Summary

- Code smells are signs that the code might be turning a bad code and likely to introduce bugs in near future
  - Refactor code continuously and not at the end of the project
  - Reusable, flexible and maintainable code
- Removing one code smell may introduce other code smells
  - Data class and data clumps
- The presence of code smell is not conclusively a sign of bad code
  - Depends on application, programming language etc.

SEI CERT Java Coding Standard

- A set of rules which are meant to provide normative requirements for code.
- Each rule is associated with a metrics for severity (low, medium, and high), likelihood (unlikely, probably, and likely) and remediation cost (high, medium, and low).
- Conformance to the rule can be determined through automated analysis (either static or dynamic), formal methods, or manual inspection techniques.

Java Coding Standard

156 rules

https://www.cert.org/secure-coding/

More than those of C/C++

SEI CERT Java Coding Standard

- Input valuation and data sanitization
- Object-orientation
- Locking and thread-safety (in concurrency lectures)
- Visibility and atomicity (in concurrency lectures)

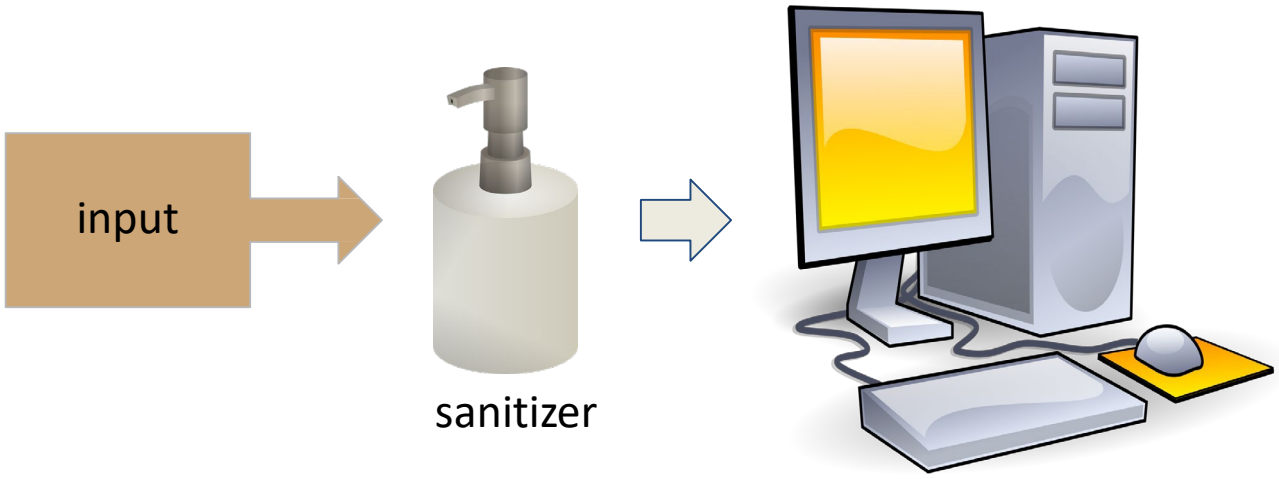
Input Validation and Data Sanitization

Many programs accept untrusted data originating from unvalidated users, network connections, and other untrusted sources and then pass the (modified or unmodified) data across a trust boundary to a different trusted domain. Such data must be sanitized.

- IDS00-J. Sanitize untrusted data passed across a trust boundary
- IDS01-J. Normalize strings before validating them
- IDS11-J. Eliminate non-character code points before validation

IDS00-J

Sanitize untrusted data passed across a trust boundary



SQL Injection in Java

```
public Connection getConnection() throws
SQLException
{
   DriverManager.registerDriver(new
   com.microsoft.sqlserver.jdbc.SQLServerDriver());
   String dbConnection =
   PropertyManager.getProperty("db.connection");
   return
   DriverManager.getConnection(dbConnection);
}

String hashPassword(char[] password) {
   // create hash of password
}
```

```
public void doPrivilegedAction( String username, char[]
password) throws SQLException {
   Connection connection = getConnection();
   if (connection == null) { // handle error }
   try {
         String pwd = hashPassword(password);
         String sqlString = "SELECT * FROM db_user WHERE
         username = '"+ username +"' AND password = '" +
         pwd + "'";
         Statement stmt = connection.createStatement();
         ResultSet rs = stmt.executeQuery(sqlString);
         if (!rs.next()) {
            throw new SecurityException("incorrect");
         }
   } finally {
   try { connection.close(); }
   catch (SQLException x) { }
}
```

See any problem?

input password does not go directly into sql query.
Hashing is a kind of input sanitation
SQL injection (1 or 3) can happen
SQL must think the whole thing is the user name. use escape via double quotes

Unvalidated Input: SQL Injection

An SQL injection vulnerability arises when the original SQL query can be altered to form a different query.

An SQL command to authenticate a user might take the form:

```
SELECT * FROM db_user WHERE username='<USERNAME>'
AND  password='<PASSWORD>'
```

If it returns any records, the username and password are valid.

The user inputs the following username:

validuser' OR '0'='0

, where *validuser* is a valid

username. The SQL command

becomes

SELECT * FROM db_user WHERE username='*validuser' OR ('0'='0* AND password=<PASSWORD>)

With a valid username and any password, the user will be authenticated.

Unvalidated Input: SQL Injection

The user inputs the following password:

' OR 0='0

The SQL command becomes

SELECT * FROM db_user WHERE username='' AND password='' *OR 0='0*

The user will be authenticated with any username.

IDS00-J

Compliant Code Example:
SQLInjectionCompliant.java

- To prevent SQL injection, use class *PreparedStatement* and method *setString/setInt* in *java.sql*.
- No SQL injection vulnerability for JRE reported in CVE since 2010 https://www.cvedetails.com/product/19 117/Oracle-JRE.html?vendor_id=93

XML Injection

Consider an online store which stores purchase orders in XML.

User selects “iPhone X” and inputs “1” for quantity.

The following order is generated.

```
<item>
<description>iPhone X</description>
<price>999.0</price>
<quantity>1</quantity>
</item>
```

XML Injection

User selects “iPhone X” and inputs “1</quantity><price>1.0</price><quantity>1” for quantity.

The following order is generated.

```
<item>
<description>iPhone X</description>
<price>999.0</price>
<quantity>1</quantity><price>1.0</price><quantity>1</quantity>
</item>
```

XML (SAX) parser (org.xml.sax and javax.xml.parsers. SAXParser) interprets the XML such that the second price field overrides the first, leaving the price of the item as $1.

IDS00-J

Sanitize untrusted data passed across a trust boundary

Noncompliant code example

```
private void createXMLStream(BufferedOutputStream outStream, String quantity) throws
                IOException {
        String xmlString;
        xmlString = "<item>\n<description>Widget</description>\n" + "<price>500.0</price>\n" +
                        "<quantity>" + quantity + "</quantity></item>";
        outStream.write(xmlString.getBytes());
        outStream.flush();
    }
```

What if a user inputs
“1</quantity><price>1.0</price><quantity>1” for quantity?

IDS00-J

Sanitize untrusted data passed across a trust boundary

Compliant code example

```
private void createXMLStream(BufferedOutputStream outStream, String quantity) throws
                IOException {
        // Write XML string if quantity contains numbers only.
        // Blacklisting of invalid characters can be performed in conjunction.
        if (!Pattern.matches("[0-9]+", quantity)) { /*Format violation*/ }
        String xmlString = "<item>\n<description>Widget</description>\n" + "<price>500</price>\n" +
            "<quantity>" + quantity + "</quantity></item>";
        outStream.write(xmlString.getBytes());
        outStream.flush();
}
```

XSS Injection: Cross Site Scripting

The website serves HTML pages to users who request them.

The attacker is a malicious user of the website who intends to launch an attack on the victim by exploiting an XSS vulnerability in the website.

The victim is a normal user of the website who requests pages from it using his browser.

The website

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

The attacker

```
<html>
Latest comment:
<script>...</script>
</html>
```

IDIDS01-J

- One strategy for avoiding XSS may include forbidding <script> tags in inputs.
- It is insufficient for complete input validation and sanitization, as the “sanitized” input may go through normalization afterwards.

Non-compliant code example

```
Pattern pattern = Pattern.compile("[<>]");
Matcher matcher = pattern.matcher(in);
if (matcher.find()) {
        // Found black listed tag
        throw new IllegalStateException();
} else {
        // . . .
}
// Normalize
in = Normalizer.normalize(in, Form.NFKC);
```

What if the input is "\uFE64" + "script" + "\uFE65"?

IDIDS01-J

Normalize strings before validating them

Compliant code example

```
in = Normalizer.normalize(in, Form.NFKC);

Pattern pattern = Pattern.compile("[<>]");
Matcher matcher = pattern.matcher(in);
if (matcher.find()) {
        // Found black listed tag
        throw new IllegalStateException();
} else {
        // . . .
}
```

All of these apply to JavaScript for DOM-based XSS as well.

Cohort Exercise 2

The program on the right is supposed to sanitize an input (to avoid XSS).

Explain why it is problematic and fix it.

Sample: <script> alert('...') </script>

Non-compliant code example

```
s = Normalizer.normalize(s, Form.NFKC);
Pattern pattern = Pattern.compile("<script>");
Matcher matcher = pattern.matcher(s);
if (matcher.find()) {
        System.out.println("blacklisted tag");
} else {
        // . . .
}
// Deletes all non-valid characters
s = s.replaceAll("[^\\p{ASCII}]", "");
```

Cohort Exercise 11:

No, if we put in non-ASCII characters in between `<script>`, it would pass the pattern check, but after that when we replace all non-ASCII charcters to "", we would get back the `<script>`.

For example, we can pass in the String s like the following:

String s = "<scri¥pt>";

Object Orientation

Object-orientation allows us to encapsulate data and preserve invariants (sometimes security policies) among class members.

- OBJ00-J. Limit extensibility of classes and methods with invariants to trusted subclasses only
- OBJ01-J. Declare data members as private and provide accessible wrapper methods
- OBJ02-J. Preserve dependencies in subclasses when changing superclasses
- OBJ04-J. Provide mutable classes with copy functionality to safely allow passing instances to untrusted code
- OBJ05-J. Defensively copy private mutable class members before returning their references
- OBJ10-J. Do not use public static non-final variables

OBJ01-J.

Declare data members as private and provide accessible wrapper methods

- Data members of a class must be declared private.
- Using wrapper methods enables appropriate monitoring and control of the modification of data members.

Non-compliant code example

```
public class Widget {
   public int total;
   void add() {
       if (total < Integer.MAX_VALUE) {total++;} else {
                   throw new Exception("Overflow");
       }
   }
   void remove() {
       if (total > 0) { total--;}
       else { throw new Exception("Overflow");
       }
   }
}
```

What is the problem?

OBJ01-J.

Declare data members as private and provide accessible wrapper methods

- Data members of a class must be declared private.
- Using wrapper methods enables appropriate monitoring and control of the modification of data members.

Compliant code example

```
public class Widget {
   private int total; // Declared private
   public int getTotal () {
        return total;
   }

   void add() {
        if (total < Integer.MAX_VALUE) {total++;} else {
                throw new Exception("Overflow");
        }
   }
  ...
}
```

OBJ04-J

Provide mutable classes with copy functionality to safely allow passing instances to untrusted code

Mutable classes allow code external to the class to alter their instance or class fields. Provide means for creating copies of mutable classes so that disposable instances of such classes can be passed to untrusted code.

What if a malicious users uses this class?

Noncompliant code example

```
public final class MutableClass {
        private Date date;
        public MutableClass(Date d) {
                this.date = d;
        }
        public void setDate(Date d) {
                this.date = d;
        }
        public Date getDate() {
                return date;
        }
}
```

OBJ04-J

Provide mutable classes with copy functionality to safely allow passing instances to untrusted code

Mutable classes allow code external to the class to alter their instance or class fields. Provide means for creating copies of mutable classes so that disposable instances of such classes can be passed to untrusted code.

Encapsulate the data!

Compliant code example

```
public final class MutableClass {
    private final Date date;
    public MutableClass(MutableClass mc) {
      this.date = new
      Date(mc.date.getTime());
    }
    public MutableClass(Date d) {
        this.date = new Date(d.getTime());
    }
    public Date getDate() {
        return (Date) date.clone();
    }
}
```

Cohort Exercise 3

The program on the right declares the variable hm to be public, so that the variable could be read by other programs. However, for data encapsulation, we do not want the content of the map to be modified, other than by methods in the class. It is argued that since hm is declared as final, it should be safe.

Explain why it is not safe and implement a fix.

Non-compliant code example

```
public class myClass {
        public static final HashMap<Integer, String>
hm
                = new HashMap<Integer, String>();

        //methods are skipped
 }
```

its a HASHMAP!!!
so even though it is final, only the reference is final.
You can still modify contents of the hash map.

To fix:
we clone

```
public static HashMap<Interger,String> getHM(){
    return new HashMap<Integer, String>(hm);
```

OBJ02-J.

Preserve dependencies in subclasses when changing superclasses

When developers modify a superclass (during maintenance, for example), the developer must ensure that changes in superclasses preserve all the program invariants on which the subclasses depend.

Noncompliant code example

NoncompliantOBJ02.java

Security is delegated to the subclass BankAccount. The client application is required to use BankAccount because it contains the security mechanism.

Later, the maintainer of the class Account added a new method called overdraft(). The BankAccount class maintainer was unaware of the change.

Consequently, the overdraft() method could be invoked directly on a BankAccount object, avoiding the security checks that should have been present.

Cohort Exercise 4

This program overrides the methods after() and compareTo() of java.util.Calendar. The programmer wishes to extend this functionality so that the after() method returns true even when the two objects represent the same date. The programmer also overrides the method compareTo() to provide a “comparisons by day” option.

Explain what could be the problem and fix the problem.

Code example

exercise4.java

Hint: Do not inherit CalendarSubclass from Calendar