

Colby



Colby College
Digital Commons @ Colby

Honors Theses

Student Research

2005

JeSS – a Java Security Scanner for Eclipse

Russell Spitler
Colby College

Follow this and additional works at: <https://digitalcommons.colby.edu/honorstheses>



Part of the [Databases and Information Systems Commons](#), [Other Computer Engineering Commons](#), [Programming Languages and Compilers Commons](#), and the [Systems Architecture Commons](#)

Colby College theses are protected by copyright. They may be viewed or downloaded from this site for the purposes of research and scholarship. Reproduction or distribution for commercial purposes is prohibited without written permission of the author.

Recommended Citation

Spitler, Russell, "JeSS – a Java Security Scanner for Eclipse" (2005). *Honors Theses*. Paper 567.
<https://digitalcommons.colby.edu/honorstheses/567>

This Honors Thesis (Open Access) is brought to you for free and open access by the Student Research at Digital Commons @ Colby. It has been accepted for inclusion in Honors Theses by an authorized administrator of Digital Commons @ Colby.

JeSS – a Java Security Scanner for Eclipse

Russell Spitler

Senior Honors Thesis
Spring 2005

Colby College
Department of Computer Science
Advisor: Dale Skrien

Contents

Chapter 1	Introduction	1
Chapter 2	Secure Coding and Java Security	
	2.1 – <i>Secure Coding</i>	3
	2.2 – <i>Java Security</i>	7
Chapter 3	Java Security Holes	
	3.1 – <i>Don't depend on initialization</i>	13
	3.2 – <i>Make everything final</i>	14
	3.3 – <i>Make your code unserializable and undeserializable</i>	16
	3.4 – <i>Make your class non-Cloneable</i>	19
	3.5 – <i>Don't rely on protected or package scope</i>	20
	3.6 – <i>Don't use inner classes</i>	23
	3.7 – <i>Make everything as private as possible</i>	25
	3.8 – <i>Sign as little of your code as possible</i>	26
	3.9 – <i>Encrypt XML generation</i>	28
	3.10 – <i>Check execution of JNI code</i>	30
	3.11 – <i>Catch all Exceptions</i>	31
	3.12 – <i>What We Can Do</i>	33
Chapter 4	Tools and Methodologies behind JeSS	
	4.1 – <i>Evolution of JeSS</i>	35
	4.2 – <i>Source Code Analysis</i>	36
	4.3 – <i>User Interface</i>	39
	4.4 – <i>The Eclipse IDE</i>	41
Chapter 5	JeSS Details	
	5.1 – <i>Goals of JeSS</i>	43
	5.2 – <i>Layout of the Eclipse IDE</i>	44
	5.3 – <i>User Interaction with JeSS</i>	45
	5.4 – <i>Source Code Auditor</i>	48
	5.5 – <i>Current JeSS Scans</i>	50
	5.6 – <i>Extending the JeSS Scans</i>	51
	5.7 – <i>Extending the JeSS Plug-in</i>	52
Chapter 6	Results of the JeSS Project	
	6.1 – <i>Goals Met</i>	55
	6.2 – <i>Deployment of JeSS</i>	56
	6.3 – <i>Future Work on JeSS</i>	57
Chapter 7	References	59
Appendix A	JeSS Users Manual	63
Appendix B	JeSS README	73
Appendix C	JeSS JavaDoc	75

1. Introduction

As we expand the application of computer technology in our society, security becomes more and more of a concern. We enter sensitive information into computers daily and assume that it is handled in an appropriate manner. This assumption is largely unfounded. Many of the applications that are in use today can be compromised and exploited. The development of secure software is necessary for the continued expansion of computer use. The process of developing this secure software has to be done on many levels. Security needs to be kept in mind when deploying and using software, but even more importantly during the development process. If software is developed with security in mind then it is far less likely to be compromised when it is deployed. These concerns apply to all software, not just applications that deal with sensitive information. Any exploitable program is a threat to a systems security. A completely innocuous program could be used to escalate user privileges and provide access to a machine. Just as a chain is only as strong as its weakest link, a computer is only as secure as its weakest program. Secure software is the responsibility of every developer.¹

In order to help a developer with this responsibility there are many automated source code security auditors. These tools perform a variety of functions, from finding calls to insecure functions to poorly generated random numbers.² These programs have existed for years and perform the security audit with varying degrees of success. Largely missing in the world of programming is such a security auditor for the Java programming language. Currently, Fortify Software produces the only Java source code security auditor; this is a commercially available package. This void is what inspired JeSS, Java

¹Gary McGraw “Software Security” [9]

² Tech FAQ [27]

Security Scanner for eclipse. JeSS is an open source, extensible program that statically analyzes source code for possible security bugs.

To tightly couple JeSS with the software development process, JeSS was developed as a plug-in for the Eclipse Integrated Development Environment. Eclipse is an open source Integrated Development Environment (IDE) developed by IBM. Eclipse is widely used by developers in both educational and commercial settings.³ The Eclipse IDE was picked for JeSS because of this widespread use, its publicly available source code, and easy extensibility. JeSS plugs into the eclipse user interface, using the standard widgets found in the development environment. The integration with the IDE and use of standard conventions within the environment makes JeSS a tool that is easy to use throughout the development process.

³ Eclipse Foundation [33]

2. Secure Coding and Java Security

2.1 Secure Coding

Software can be compromised in two general ways: by gaining access to proprietary code or stored data, and by using the application to escalate local privileges. Software packages are often distributed with highly specialized code. If the package is poorly designed it could be possible for an end user to gain access to the specialized code, revealing the way that it works. If a program stores sensitive information then it is important to protect the data structures used to store this information, so it is not compromised. This can happen through undocumented or improperly protected access points. A user can escalate local privileges when they “hijack” the execution of an application in order to gain the security privileges currently held by the application. In such cases the user can abuse the application’s privileges to run code they would otherwise not be able to run.

If software becomes compromised in any of these ways, it can cause serious repercussions. If the software is commercially distributed, the process of patching and updating distributed software must be undertaken to rectify the security holes. In addition, any competitive edge gained by proprietary code is likely lost. Worse, a software vendor’s reputation could be undermined if there is a major loss of sensitive data. If software is used in a privilege escalation attack then any machine that runs the software is insecure, and vulnerable to hackers. Any end user of the software will have justified concerns about running it on their machines. Developers must make efforts to develop secure software in order to prevent such problems.

An important but subtle distinction needs to be made between secure software and security software. The latter is software that is designed as a reactive solution, it is made to try and “plug” the security holes that are present in existing software. Security software identifies and prevents malicious attacks on computer systems. Virus scanners, firewalls, and input filters are examples of this type of software.⁴ Security software is made necessary because there is a lack of secure software. Secure software is designed from the ground up with the security of the application in mind. Reactive solutions to security problems are less necessary if applications are more secure in general. If the software is designed and coded with security in mind then the distribution and use of the software will be much less worrisome from a security standpoint.

The factors that need to be considered in the development of secure software are present throughout the development process. Secure software starts with the general architecture of the application. Security must also be a consideration when deciding implementation details, such as the approach and philosophy of the programming team. There must also be third party analysis of the software’s security and close observation of the software when it is “in the wild.” All of these steps can be affected by corporate policy, design requirements, compatibility requirements, and implementation constraints. A more detailed look at this process can be found in Gary McGraw’s paper “Software Security” [9].

JeSS is a tool to be used during the implementation phase of secure software development. In the implementation phase developers make a few important decisions right off the bat, such as the language and development environment for the application.

⁴ Gary McGraw “Software Security” [9]

While it is not necessary to restrict either of these choices, a single language, or IDE can help a development team retain a single, thorough security policy throughout the coding process.⁵ These choices can be determined by corporate policy. It is typical for a corporation to mandate a language or IDE to be used in development.⁶

These choices also affect the secure software development process. For example, an IDE that supports integrated refactoring can be a great asset. It is far easier to fix an identified security hole if refactoring does not have to be done by hand. Additionally, a well-designed IDE makes it easier to refactor buggy code.

The other decision in this phase of the development process concerns the actual language used to code the application. Every language has its problems when it comes to security. Some languages have more severe problems than others. The designers of a language have different goals in mind, and security is not always one of them. Regardless of the language choice, the actual coding process should be done with the security faults of the chosen language in mind.⁷ These faults embody themselves in many ways. A fault can be as simple as a call to an insecure function (such as `strcpy` in C)⁸ or as apparently innocuous as the use of certain coding structures (such as inner classes in Java).⁹ Programmers should be versed in the basic strategies for avoiding problematic code. However, it is not thorough enough to rely on the programmer, who may introduce many security bugs through oversights and omissions, despite the best of intentions. An automated tool should be used to ensure that the code is free of any security problems

⁵ Gutschmidt “Securing Java Code” [4]

⁶ Susan Kohler GE Medical Systems (Personal Conversation)

⁷ Gary McGraw “Software Security” [9]

⁸ John Viega “its4: A Static Vulnerability Scanner for C and C++” [13]

⁹ Gary McGraw “Twelve rules for developing more secure Java code” [2]

that can be statically detected. This is the essence of JeSS. It is an automated tool to find statically detectable security bugs in Java. There are many such tools for C and C++¹⁰, but there is currently only one available security auditor for Java.¹¹

In C and C++ there are many security problems that are statically detectable. There are quite a few built in functions that can lead to buffer overflow errors and others that lead to race conditions.¹² There are many publicly available scanners that will analyze your source code to find these dangerous calls; a very thorough C and C++ scanner is detailed in [13].

Currently Fortify Software is the only company that makes a security auditor for Java source code. Fortify supplies complete security analysis of applications. Their services are available on a subscription basis, and they consist of two parts: source analysis and attack simulation. In the source analysis Fortify provides tools that ensures your code adheres to the guidelines of secure coding and identifies problematic sections of your code.¹³ The lack of security auditors made for Java can possibly be attributed to a few reasons. First of all, Java is a relatively young language, and its widespread use in industrial software development is only beginning. Without a history of industrial use, tools that ensure industrial strength code are largely missing. Another consideration is the lack of education in the security concerns associated with Java. As Java has been hyped as a secure language without the highly publicized shortcomings of C or C++, it is assumed that no special precautions need to be taken to ensure the security of the code.

¹⁰ Tech FAQ [27]

¹¹ Fortify Software [28]

¹² John Viega “its4: A Static Vulnerability Scanner for C and C++” [13]

¹³ Fortify Software [28]

The following sections provide an extensive look at the Java security architecture and the coding structures that can lead to security holes.

2.2 Java Security

The Java programming language is designed to be the ultimate portable language. Sun developed the language to live up to the ideal “write once, run anywhere.”¹⁴ To accomplish this task Java was developed as an interpreted language. The byte code of a compiled Java program cannot be run natively on most machines; rather the code is interpreted and run by a Java Virtual Machine (JVM). The JVM is a program developed to Sun’s specifications by commercial and open-source developers. The different JVM programs available are developed to run on a certain type of machine. This allows the JVM to interpret the Java byte code and execute the appropriate native machine code.¹⁵ This system allows the execution of any Java program on any machine for which there is a JVM available.

This portability has promoted Java as an embedded language in web content, and has promoted its use in many distributed applications. Foreseeing this use, Sun has built many security features into Java. These security precautions center on the concerns produced by anonymously distributed code. With embedded web content it is possible, by simply accessing a web page, for a user to download and execute Java code without ever knowing of its existence. The anonymous distribution of executables is a huge concern because the true purpose and effects of such a program are completely unknown.

¹⁴ Steven Fritzing “Java Security” [5]

¹⁵ Steven Fritzing “Java Security” [5]

To solve this problem Java uses what is known as the “Sandbox” model.¹⁶ The Sandbox model separates code run on the JVM into two general categories, trusted and un-trusted (in actual practice there are varying degrees of these two categories). This separation is done through the JVM’s Security Manager. This system limits access and privileges to untrusted code. Untrusted code is executed with different restrictions, and, as the name implies, it is not trusted to perform many basic system calls. The privileges to write to the disk, access the disk, or connect to a server are restricted in Java applications unless the JVM, via the local Security Manager, grants explicit permission for the application to perform the specific task.¹⁷ This system of security allows the user to execute anonymously distributed applications without having to worry about malicious side effects. This allows users of Java to take full advantage of the language’s portability.

Even with the use of the Sandbox model and the relative safety of running a Java program there are still security concerns in the Java language. The system of differentiating between trusted and untrusted code is a safeguard for the user, protecting them from harmful side effects, maliciously introduced or accidental. However, another real concern is the security of the application. It is a concern for any developer to ensure that their code is not accessed in an unintended way by an end-user.

For some, this concern is as basic as protecting secure information. Applications that deal with personal information such as social security numbers, health records, or even bank accounts need to make sure that an unauthorized user does not access this information. Other applications may have proprietary code that performs a task in a unique way. Preventing a competitor from accessing this is a basic business concern.

¹⁶ Steven Fritzing “Java Security” [5]

¹⁷ Steven Fritzing “Java Security” [5]

Another concern is the “hijacking” of an application. Even if an application secures no private information nor holds any critical code, it is necessary to secure the program. If the program is granted trusted privileges on a local machine it may be possible for someone to introduce additional code to escalate their own privileges on that machine using that program. In addition, concerns about the developer’s integrity come into play. It is possible, if the application is not secure, for someone to introduce malicious code into a program and then redistribute it. With this done, the application will still appear to be the original developer’s work, but will now have malicious side effects.

A brief overview of the general nature of Java’s security holes follows. Some of the holes are related to object-oriented language features in general, such as inheritance, and polymorphism. Other security holes can be attributed to features specific to the Java language.¹⁸ A later section discusses specific examples and preventative measures for these security holes.

One security concern is the existence of an object in an insecure state. Many programmers make false assumptions about what can and cannot be done in Java. This leads to incorrect assumptions about the infallibility of class invariants. This can ultimately lead to objects in states never intended by the programmer, subsequently allowing a user to gain access to the program. The existence of an object in an unanticipated state can lead to serious security breaches and even, in some instances, to unrestricted remote access to the system.¹⁹

Malicious extension is a very real concern in Java. The Java language is designed so classes can be easily extended and reused. In the local scope of an application this is a

¹⁸ Gary McGraw. Securing Java [3]

¹⁹ Gary McGraw. Securing Java [3]

concern. The extension of classes could compromise the behavior of the application. By using a mix and match attack,²⁰ the end user can introduce subclasses of their design into an application. These subclasses are completely unregulated and can include malicious code.

A program's public interface is defined by the methods and variables that are publicly available. Many programmers fail to consider the implications of the modifier they use to designate their methods. The end user can use any method designated as public. All of these methods are access points, and possible points from which a program can be compromised. Typically these access points are more numerous than the programmer originally intends. Through a variety of means even some private and protected methods can become used as a public method would.

While it is virtually impossible to make any program completely secure, certain steps can be made to limit the possible ways in which a program could be compromised. While the idea of secure software development is a relatively new topic²¹ there is some decent documentation. A comprehensive look at building secure Java software can be found in Gary McGraw's book "Securing Java: Getting Down to Business with Mobile Code" [3]. In this book Gary McGraw briefly presents twelve guidelines for producing secure Java code. The following section closely examines Gary McGraw's specific security guidelines as well others not included in his original work. This discussion includes specific exploits and then precautions that can be taken to prevent their use. All of these security holes can be classified in the general categories we saw above:

²⁰ Gary McGraw. Securing Java [3]

²¹ Gary McGraw. "Software Security" [9]

malicious extension, public interface abuse or extension, and producing objects in an illegal state.

3. Java Security Holes

3.1 Don't depend on initialization

To set up class invariants in Java it is common practice to do so in the constructor. Every variable that is needed for the program to run properly should be initialized in this process. Code written in the other public methods often relies upon the conditions initialized in the constructor. In particular, it is often assumed that these methods will never be called before the constructor is fully executed. If it is possible to run methods without the constructor setting up the crucial invariants, it opens up a possible means of attack for someone trying to access your code. As a result it is necessary to make sure that the object has been properly initialized before any method in the class is executed. This may seem like a moot point, but in Java there are a few ways to allocate objects without calling a constructor of that object.

For example, it is possible to generate a byte array that can be deserialized into a java object.²² Also, by calling an object's clone method you can possibly create a new object without calling a constructor. Furthermore with XML it is possible to "demarshall" a XML file in order to create a new instance of an object.²³ All of these methods bypass the constructor and subsequently bypass any security measures or class invariants that are set up by the constructor. As a result, the class invariants may not be set up properly and the value of any field in the object is at the discretion of the person creating the serialized byte array, the clone, or the XML file. All of the ways in which this can be done will be covered in more detail in subsequent sections of this paper.

²² Kalinovsky. Covert Java [7]

²³ Kalinovsky. Covert Java [7]

3.2 Make everything final

Method polymorphism and dynamic method invocation are building blocks of the Object-Oriented approach. Polymorphism is the system that dynamically determines the objects' type and which version of a particular method should be called depending upon how the method is overridden in the inheritance tree. For example, if a variable is declared to have the type of a certain superclass and then is initialized to refer to an object of a subclass of that superclass and if the method `foo()` is defined in both classes, then it is polymorphism that will dynamically determine at run time that the sub-classes' version of `foo()` should be called. This is a dynamic system in that the executed method is completely determined at runtime, and there is no way for a programmer to specify the class in which the method is to be found. A method in a superclass can be overridden in a subclass if that method is not declared final.

Overriding is a system that can change the behavior of a superclass' method. To do this in Java it is necessary to create a method in the subclass that has the exact same signature as the method in the superclass. The parameters, name and return type must be identical to the method in the superclass. When this method is called on an instance of the subclass the method as defined in the subclass is called. Herein lies the vulnerability. The only requirement of the overriding method is that it has an identical signature. The additional code executed is completely defined by the author of the subclass. While the signatures remain the same, the author of the subclass could modify the behavior of the method and compromise the security of the class.²⁴

²⁴ John Viega. "Statically Scanning Java Code: Finding Security Vulnerabilities" [6]

To prevent this from happening it is possible to use the modifier “final” when defining variables or methods. This tags the variable so that its value cannot be changed and tags the method so it cannot be overridden in a subclass. Use of this modifier eliminates many possible applications of an extended class or even the extension of the class itself. Even though future programmers are restricted to your implementation, this is a very important precaution to take. When a method is not marked final then it can be overridden and the behavior and actions of the overridden method are completely out of your hands. It is possible to that the new method could execute code that compromises the security of your application.

An example of such a security bug is as follows. Let us say a method of a class is set up to check whether the passed username and password is a valid match.

```
Class PassCheck {
    HashMap users; // key = username, value = password
    ...
    public boolean passwordCheck(String username, String
        password){

        return password.equals(users.get(username));
    }
}
```

Exploiting the fact that neither the class nor the method is final, a subclass could be defined as follows:

```
Class BadPassCheck extends PassCheck {
    HashMap stolenUsernames;
    ...
    public boolean passwordCheck( String username, String
        password) {
        boolean isValid = super.passwordCheck(username,
            password);
        if(isValid)
            stolenUsernames.put(username, password);
        return isValid;
    }
}
```

In this example the subclass behaves just as its superclass implementation would, except that it also stores the passed values in a new `HashMap` that the author of the subclass could access in any number of ways. This is an overly simplified example; hopefully more security precautions would be taken to secure usernames and passwords. The purpose is to highlight the fact that a method can be overridden to include malicious code without making any behavioral changes. Another example might include getter and setter methods that are not finalized; similar to this example, the values passed in may be stored and later analyzed by the author of the subclass.

Extensibility is one of the powerful features of a language with inheritance; with as many approaches as there are programmers, classes can be extended in infinitely many novel ways. This is a plus except when dealing with applications where security is a concern. Any un-finalized class or method can be extended in unforeseen ways and executed in other parts of your code through polymorphism. It is far easier to approach the reciprocal to this problem and finalize everything except for the few cases where it is necessary not to do so.

3.3 Make your code unserializable and undeserializable

Serializing is an outdated way of storing the state of an object. The current Java documentation encourages the use of XML as a replacement.²⁵ However, since serialization may still be implemented as a way to retain backwards compatibility it is a valid issue to discuss. Serializing breaks your code into an external byte array that stores

²⁵ Sun Microsystems. Java Architecture for XML Binding. [24]

the state of the variables in the class. This byte array stores only the necessary information for reconstructing an instance of the object in the same state.²⁶

Security problems arise because the object is stored in an unprotected byte array. This byte array can then be examined by anyone who has access to the disk space that is being used to store it in. In the case where an application is being distributed, this is the hard disk of the end user. Even if the storage space of the array is inaccessible it is possible for the output stream of the serialization to be monitored.²⁷ With access to the serialized object in its byte array form, it is possible to ascertain the values of any of the variables. All public and private variables can be examined. If one of these variables is a reference to another object that is serializable, all of that object's fields are also stored in the byte array.

To prevent this problem, the class must be designated as unserializable. The solution of not implementing the serializable interface is not sufficient, as a subclass could implement it. To properly prevent the serialization of your code you must implement the serializable interface and have the requisite `writeObject()` method throw a `class nonserializable exception`.²⁸ Code for this follows:

```
public final void writeObject(ObjectOutputStream out)
    throws java.io.IOException {
    throw new java.io.IOException("Object cannot be
    serialized");
}
```

In a related problem it is important for a class to be undeserializable as well. One deserializes an object, by calling the `readObject()` method as defined in the serializable

²⁶ Sun Microsystems. *Serialization Specification*. [18]

²⁷ Sun Microsystems. *Security in Object Serialization*. [17]

²⁸ Gary McGraw. *Securing Java*. [3]

interface. This takes a byte array as generated by the `writeObject()` method and translates it back into an object of the stored state. This is necessary if your class is serializable in that it must be possible to return from the byte array for serialization to be a useful feature.

This process also causes some security concerns. There is no way to ensure that the byte array that you are deserializing contains a copy of an object that was previously instantiated. It is possible for someone to generate a byte array that is a valid serialization of an object of your class, that contains any values that the creator wishes for any of the variables. This gives the creator of the byte array complete control of the state of the object, making it possible for them to create an object in an insecure state. In addition it is possible to use this method to create as many instances of a particular class as desired. This is a security problem if the class is used in a way where its uniqueness is assumed. Such an example would be a duplicate security manager or multiple username/password databases.

This problem can be solved in a nature similar to the serialization problem. It is inadequate simply not to implement the `Serializable` interface and assume the problem is solved. You must also throw an exception when deserialization is attempted.²⁹ Code for this follows:

```
private final void readObject(ObjectInputStream in)
    throws java.io.IOException {
    throw new java.io.IOException("Object cannot be
    deserialized");
}
```

²⁹ Gary McGraw. *Securing Java*. [3]

3.4 Make your class non-Cloneable

Cloning is a way to obtain an object that is in the same state as the source. To initialize an object using the clone() method the result should be an object with instance variable values identical to the values of the source's instance variable values. This is a quick and easy way to duplicate objects without having to manually replicate the values of the class's variables. This method can be very useful in a variety of applications.³⁰

The problem with cloning is that a typical clone method will produce a shallow clone, one that simply has references to the variables in the original object. This is a problem from a design standpoint because changes in one object could affect the values of the other. Even if the clone method is set up to create a deep clone by cloning all the internal variables and using those to create the new object, there is a problem. The problem is that it is creating a new object of a class without calling one of its constructors. This sidesteps any security measures that you have built into the original constructors. Any security precautions that you may have set up or restrictions on the number of objects of a certain class are completely irrelevant. While the class will have a valid internal state it is possible that the existence of a new object will violate security measures that have been set up at the application level.³¹

This problem is solved in a way similar to the way the serialization problem was solved. Since it is possible to extend your class and add previously unimplemented interfaces it is possible that a clone method could be defined even if you don't do it yourself. To prevent this possibility from happening the clone interface must be

³⁰ Sun Microsystems. Cloning Objects. [26]

³¹ Sun Microsystems. Cloning Objects. [26]

implemented and the clone() method must explicitly throw an exception. The following code will do such a thing:³²

```
public void final clone() throws
java.lang.CloneNotSupportedException {
    throw new java.lang.CloneNotSupportedException();
}
```

3.5 Don't rely on protected or package scope

It is important not to rely on package protection to secure your code. It is a common misconception to think that only classes in your package, as you define your package, can access package or protected methods and variables. There are a number of very simple ways to work around this protection. This is a security hole that may be caused by a lapse in attention in that any variables or methods that are not designated with either public or private are automatically designated as package visible at compile time.³³ A variable or method designated as having package scope means that any class within the package can access it and so can execute or modify it. This seems like a reasonable option because it allows the classes of your application to call methods and manipulate variables, but prevents classes that are not within your application from getting such access. It is a feature created for internal communication within an application.

This feature, however, is a security shortcoming of Java. It is impossible to absolutely define which classes are members of your package, in that there is no way to seal your package without the possibility of another class being added to it. If another

³² Gary McGraw. Securing Java. [3]

³³ Kalinovsky. Covert Java. [7]

class were to be added then the new class would have access to all of the package protected variables and methods in the package. Because of this, it is essential to make sure that all package protected variables and methods are not ones that could compromise the security of your application. It is especially important to scan for this security hole because of Java's default designation of package visible to untagged variables and methods. If the programmer forgets to designate a variable, the program will run normally and the missing designation will not be caught at compile time. This missing designation could cause a serious vulnerability in the application.³⁴

An example of such flawed code is as follows:

```
public class Course {
    Vector students = new Vector();
    ...
    public addStudent(Student o) {
        students.addElement(o);
    }
}
```

In this class the students in a particular course are stored in a vector labeled “students.” This variable is designated to have package since it is not explicitly designated as protected, public or private. With its package designation the vector is available to any class in the package, or any class later added to the package. This unrestricted access would allow any other class to access or modify the vector. Whether this is due to the implementation of the class or because the author simply forgot to specify it, it makes the variable “students” accessible to all other classes with the same package heading. As we have seen before having the same package heading does not ensure that it is a trusted class.

³⁴ Kalinovsky. Covert Java. [7]

There are two easy ways to access package protected members of a class. Which option you use depends upon the precautions the authors of the code take. If they simply distribute the code in a JAR file without sealing it, it is simply a matter of defining a new class with an identical package declaration to the package you wish to gain access to. For example if you are trying to gain access to edu.colby.JeSS you simply need to declare a new class with the header “package edu.colby.JeSS;” you then must modify the CLASSPATH so it includes the directory where your additional class is stored before it loads the original JAR file.³⁵

If the package is distributed as a sealed JAR, then the process involves one more trivial step. A sealed package sets the precedent that all classes that are loaded must come from the same JAR file, making the technique previously described ineffective. However, unsealing a file is easier than adding a new class. To unseal the file you simply have to modify the JAR’s manifest file and change the Boolean value of the Sealed attribute, or extract the JAR and recompile it with the new class in the same directory as the other classes.

3.6 Don’t use inner classes

The use of inner classes is a convenient way to implement many features of Java. Inner classes have the benefits of unrestricted access to the enclosing class, including access to all variables and methods, public or private. It is also beneficial from a design point of view in order to decrease the number of classes in the developer’s perspective, in that it allows you to avoid cluttering your package with classes that are only a few lines

³⁵ Kalinovsky. Covert Java. [7]

long. One common use of inner classes is in the implementation of `ActionListeners`. These are small classes that typically have only one method. It is beneficial to make the listener an inner class so you can access the variables of the containing class without restriction. This unrestrictive access can greatly enhance the utility of the listener. There are many other cases where an inner class is used. Whether for simplicity's sake or for design purposes inner classes are a commonly used feature in Java.

However, the implementation of inner classes in the Java byte code causes a large problem. In Java 1.0 there were no inner classes; all classes had to be defined separately. Therefore all classes were compiled independently of one another and stored as such in their compiled form. In order to preserve backwards compatibility Java compilers have to compile inner classes into separate class files, effectively making inner classes no different from any other class in the package. This causes two major problems. First, the inner class has to be able to access the enclosing class's methods and variables. This includes any private methods and variables in the enclosing class, so special precautions need to be made to protect this right. In addition the inner class is now a high level class so any other class in the package can access it, not just the enclosing type. Because the compiler has separated the inner class from its enclosing type during compilation, any class in the package can now access the public and protected methods and variables in the inner class. This now places the inner class under the protection of the package scope, which is problematic due to a mix and match attack as discussed earlier. In addition, since the inner class is now separate from the enclosing class, the special access to the variables and methods of the enclosing class needs to be accounted for. In order to preserve this access the compiler uses a dangerous trick: it modifies the declared

protection of the methods and variables in the enclosing class. No matter whether a variable or method was declared private, the compiler designates it as package visible, so the former inner class still has access to the variable.^{36 37}

This vulnerability is especially dangerous because of its dual implications. Simply having an inner class, in any context, compromises all of the code in both the inner class and the enclosing class. All variables and methods, no matter their designation, are made to be package accessible. Due to the shortcomings of the package scope these variables are essentially publicly available. In addition, all of the methods in the inner class are also package accessible, available to all classes in the package, not just the enclosing class. An example of a security hole that can arise because of these shortcomings is as follows:

```

Class BankAccount {
    private Integer accountNumber;
    private AccountHolder personalInfo;
    private AccountManager observer;
    ...
    public BankAccount( Integer accountNumber,
        AccountHolder person){
        this.accountNumber = accountNumber;
        this.personalInfo = person;
        personalInfo.addPropertyChangeListener(new
            PropertyChangeListener() {
                public void propertyChange(
                    PropertyChangeEvent e) {
                    String prop = e.getPropertyName();
                    observer.accountAction(prop,
                        accountNumber);
                }
            });
    }
}

```

³⁶ Kalinovsky. Covert Java. [7]

³⁷ Gary McGraw. "Privileged Code in Java" [8]

This example would be compiled into two separate classes, making all of the variables of the class `BankAccount` accessible to any class in the package. In addition a class added to the package could call the `propertyChange` method, pass in their own `PropertyChangeEvent` and maliciously change the class invariant.

3.7 Make everything as private as possible

All variables, methods and classes should be designated as private unless there is a very good reason for them not to be. A variable, method or class that is designated as public can be executed or accessed by any class at any time. Every public variable, class or method is another way for someone to access your code. Closing these doors is essential to securing your code. These access points are essential for a functional application but they should be severely limited. Only methods, and classes that are essential for the codes' public interface should be designated as public.³⁸ All variables, except in rare circumstances, should be designated as private and should be accessed through getter and setter methods. This will prevent access to variables that should not be accessed externally. If properly written these methods can also prevent unauthorized classes from accessing the variables. Another advantage of a setter method is that the author of the class can ensure the variable is in a valid state by screening the passed value before the variable is set.³⁹

³⁸ Gary McGraw. Securing Java. [3]

³⁹ Skrien. Intro to OOD. [12]

3.8 Sign as little of your code as possible

Java introduced digital signatures as a way to verify the authenticity of publicly distributed code. Through a few different methods, a digital signature attached to an archive or JAR is supposed to authenticate the code it is attached to. The signature is then compared to a database of signatures to ensure that the code was originally distributed by a source that you trust. This system of authentication is necessary for mobile code. Most programs will have to execute some privileged operations in order to accomplish their designed task. These privileged operations are as simple as reading or writing to disk or creating new network connections. To grant these privileges there must be a formal mechanism to authenticate the source of the code. Signed code is Java's answer. When Java executes code that is signed it first verifies the signature against a personal, or public database (VeriSign for example). After verification the Java VM will then grant the privileges the end user has set for that signature source.⁴⁰

Code that is signed will then have all privileges that the user believes the source is entitled to. By using a digital signature as a "stamp of approval" a code developer is vouching for all the code within that JAR. Hopefully there will not be any security holes anywhere in that JAR that will allow access to outside users. However, if there are compromising holes, all of the privileges granted by using the digital signature will then be available to that outside user.⁴¹

It is extremely likely that some code in any application will have to use privileged operations at some point in order to do its job. These privileges must be granted in a reasonable, secure manner. A digital signature is such a reasonable process. The

⁴⁰ Sun Microsystems. Security and the Java Platform. [16]

⁴¹ Gary McGraw. Privileged Code in Java. [8]

privileges granted should be restricted as much as possible. Only the blocks of code that are executing privileged operations should be tagged as privileged. This is possible using the `doPrivileged` API. This allows the programmer to grant privileges only to certain sections of their code, revoking those privileges when the block is complete. One note on this matter is that Java encourages the use of inner classes to implement this code. As explained earlier this should be avoided. It is possible to implement the `doPrivileged` API without inner classes and this should be done to ensure security. An example of a privileged code block without using anonymous classes is as follows:⁴²

```
class MyClass{
    Public void foo() {
        ...
        //create a new object with the secure action
        //detailed inside
        SecureAction secure = new SecureAction();
        //execute the secure action as defined in the
        //run() method of
        //SecureAction
        AccessController.doPrivileged(secure);
        ...
    }
}
class SecureAction implements PrivilegedAction {
    public SecureAction{};
    public Object run() {
        //this is the privileged code
        return System.getSecurityManager();
    }
}
```

In this example `MyClass` does not have the privileges to access the security manager. To do so, privileged code is called to gain access. This set-up limits the privileges of accessing the security manager to precisely where that privilege is needed. This prevents any code in the class from abusing this privilege, specifically, methods that may have been compromised due to other security holes.

⁴² Gary McGraw. Privileged Code in Java. [8]

3.9 Encrypt XML generation

Extensible Markup Language (XML) is the preferred means of Java object storage. It has replaced the deprecated Serialization and Deserialization API's and is currently the supported method of storing instances of objects. XML is a markup language that uses tags of the form `<...>`, `</...>` to open and close information fields within an XML document. The Java Architecture for XML Binding (JAXB)⁴³ presents the standard method for storing instances of objects (“marshalling”) in XML and creating instances of objects (“demarshalling”) from an XML document. The classes that perform the Marshalling and Demarshalling can be automatically generated from an XML schema document. An XML schema document dictates the form of the tags and how the information in a given object is to be stored in XML. With the classes that are generated from this document one can archive an instance of a class into an XML document. The reverse of this process is also possible. Given an XML document in the proper form an instance of a class can be created as dictated by the values of the XML fields.

This system generates the same security problems as serialization. However, the simplicity of the XML language and range of its use lead to even easier abuse. In serialization the information was stored in a binary array. The information was very accessible but it still had to be translated from binary into usable form. In XML information about an object is stored in a very easy to read markup language. The nature of the language dictates the presence of tags that define what the enclosed data refers to.

⁴³ Sun Microsystems. Java Architecture for XML Binding [24]

There is no longer the extra step of decoding the byte array generated by serialization; the information is stored in plain text.

The parallel problem with deserialization also exists. By the nature of XML, generating documents in XML form is very simple. It is only a matter of generating the proper tags in the right order. Once this format has been discovered (or simply obtained from an XML schema document) it is very easy to generate an “archived” version of an object with the stored variable values of your choosing. This XML document can then be used to introduce an object in an insecure state back into an application through the demarshalling process. It would also be possible to introduce multiple instances of an object whose uniqueness is essential to the integrity or security of the application. Such an example would be a duplicate security manager or multiple username/password databases.⁴⁴

The solution to this problem is not a trivial one if one wants to use XML as a way to archive files. If this is to be done, then the output stream of the XML generation must be stored in an encrypted format. This is done by creating another XML document that specifies the encryption type and then stores the encrypted data, without reference to what type of information is stored within the document. The process of encrypting and decrypting data will slow down the processing time for both the archiving of objects and the subsequent retrieval, making it a costly option.⁴⁵

⁴⁴ Kalinovsky. Covert Java. [7]

⁴⁵ Sun Microsystems. Java Architecture for XML Binding. [24]

3.10 Check execution of JNI code

One of the main objectives of object-oriented programming is the idea of code reuse. One should never have to write the same code twice.⁴⁶ When all the code is written in Java it is a trivial matter to incorporate these blocks of code. However, if the existing code is written in another language there is an impasse. This is where the Java Native Interface comes into play. The JNI allows the programmer to run libraries or applications written in another language. This interface creates a seamless integration of the code into an application. Both the Java code and the native code can call each other's methods, utilize the other's objects, and the native code can even perform Java-specific functions such as throwing exceptions. This integration is a tool designed to allow code reuse and to improve performance of the application through execution of assembly code, or using features of a language that are not available in Java.⁴⁷

This feature of Java is a very powerful tool for a developer to use. However, the security measures built into Java are not always present in the native languages. The security precautions implemented in Java do not extend to the embedded native code. As a result the Security Manager will typically restrict the execution of native code.⁴⁸ If a program relies upon native code for successful completion then the user will have to allow the execution of the native code embedded in the program. To ensure that this code is not the source of any security holes it is essential that a properly focused security scanner be used on the native code. A number of such scanners for C and C++ can be found at [25].

⁴⁶ Skrien. Intro to OOD. [12]

⁴⁷ Sun Microsystems. Overview of the JNI. [25]

⁴⁸ Sun Microsystems. Java Security Architecture 1.5 [19]

3.11 Catch all Exceptions

In Java error handling is implemented with the use of throwing and catching exceptions. Exceptions are thrown when illegal events occur or unstable states of objects are formed. These exceptions are either thrown by the system or by code written into the application. The exceptions store information about the error and return through the call stack until they are caught in a try-catch block. If these exceptions are not caught then they will cause the JRE to shutdown, terminating the execution of the application. When exceptions are thrown, the normal execution of code is disrupted, and the exception is returned up the call stack. By utilizing this system the programmer can handle errors, such as improper array access, inability to access a certain file, or improper privileges, with a predetermined block of code. With a properly placed catch block, whenever an exception is thrown in the subsequent call stack the catch block can handle that exception.⁴⁹

Exception handling is a powerful tool that allows the programmer to handle errors in a predetermined way. However this can also be a problem. If an exception is caught and is not acted upon then the events that led up to the illegal action or object state will remain unreported and unresolved. This can result in one of the objects in the application existing in an illegal or insecure state. By not acting on the error messages passed on by the exception, a program has the possibility of having objects exist in a state that was never intended by the original programmer, this state possibly being insecure. Another problem that can arise is if the catch statement is too general. If the catch statement is set up to catch all classes that are subclasses of type "Exception" then it is very possible, and

⁴⁹ Sun Microsystems. Handling Errors with Exceptions. [21]

likely, that the catch block will catch and handle exceptions that weren't originally intended to be caught. The overly general catch statement will catch any exception thrown by the subsequent call stack.⁵⁰ This includes system exceptions such as an `ArrayOutOfBoundsException` and any exceptions thrown by the security manager. These will all be handled in an identical manner, which is defined by the body of the catch block. If these exceptions are system-thrown exceptions they should not be caught and should rightly terminate the execution of the application. Improperly catching these exceptions may cause serious side effects if the program continues to run.

One particular problem of failing to catch exceptions occurs when an exception is thrown in a constructor. Since exceptions interrupt the normal flow of code any subsequent code after the exception is thrown will be left unexecuted. This is a problem if the constructor has passed a reference to itself to another object. An example of this is found in [11].

```
public class CSaver {
    public C c;
}
public class C extends ASuper {
    public C(CSaver s) throws SomeException {
        super();
        s.c = C.this;
        ...
        throw new SomeException();
        ... //more code setting up class invariants
and security measures
    }
}
```

In this example the class `C` leaked its `this` variable. The object `s` of the class `CSaver` has a reference to this variable. If the new `SomeException` is caught and not properly dealt with then it is possible for `CSaver` to use its reference to the malformed instance of

⁵⁰ Sun Microsystems. Handling Errors with Exceptions. [21]

the C class. Since the exception was thrown before the final lines of the constructor were executed, the class invariants and security precautions were not executed. This means that the instance of C as stored in the CSaver class is possibly in an insecure state. Failing to properly deal with a thrown exception has introduced an insecure object into the application.

3.12 What We Can Do

John Viega, Gary McGraw, Tom Murtsdoch, and Edward Felten first introduced the idea of a static Java scanner in their paper “Statically Scanning Java Code: Finding Security Vulnerabilities.”⁵¹ In this paper they explore the idea of creating a security auditor to statically detect the security holes as detailed by Gary McGraw and Edward Felten in their book “Securing Java: Getting down to business with mobile code.”⁵² The proposed scanner used the visitor pattern to traverse an AST and detect the security holes. However, this software was never fully developed or distributed. The program that was created did scan for the security holes detailed in the book. However it was a standalone program that was proclaimed a “hack” by the authors.⁵³ JeSS was created to finish the work first proposed in this paper. JeSS was created to be an extensible, easy to use, publicly available security auditor that can be easily integrated into the development process. JeSS was created to provide an alternative to the services sold by Fortify Software.

⁵¹ Gary McGraw. “Statically Scanning Java Code”[6]

⁵² Gary McGraw. Securing Java[3]

⁵³ Gary McGraw (Personal Correspondance)

4. Tools and Methodologies behind JeSS

4.1 Evolution of JeSS

JeSS has two major parts to it, each one requiring different decisions and tools when it came to its implementation. JeSS can be split into its users interface and the backend, the actual security auditor. The original idea behind JeSS was for it to be developed as both a standalone application and as a plug-in for Eclipse. The standalone application was to be developed using the Java Swing package. This would allow a graphical user interface that could be created from pre-defined widgets. The development of the Eclipse plug-in was originally seen as simply being able to launch the standalone application from within Eclipse, without integrating the JeSS functionality into the Eclipse platform. In this standalone application the security auditing was going to be done using the publicly licensed tools javaCC and jjTree. These tools can produce a parse tree of a Java compilation unit and inject the visitor pattern into it. Eventually this original design was scrapped and JeSS evolved into its current form; as a fully integrated Eclipse plug-in. JeSS fully integrates itself into the user interface native to Eclipse. All input and feedback is done through the standard conventions used in Eclipse. The back end also uses Eclipse packages. AST's and visitors are made using the org.eclipse.jdt.core.dom package. The following sections take a closer look at the two parts of JeSS and the tools used in their implementation.

4.2 Source Code Analysis

To perform a static analysis of source code, JeSS uses Abstract Syntax Trees⁵⁴ and the Visitor pattern.⁵⁵ The program generates an AST for the given source code and then traverses the tree using the Visitor pattern.

An Abstract Syntax Tree (AST) is a representation of a compilation unit. Using a programming language grammar as the rules for construction, an AST is a hierarchical tree that represents the given code. An AST represents code in a standard way as defined by the grammar. The rules that make up a grammar also determine the children of a node. This creates a hierarchical tree; sub-statements are grouped underneath their parents. For example, in an AST representing a Java “for” loop, the children of the “for” node would be the loop conditions and the block statement to be executed. There are two major advantages to representing source code in such a structure. First of all, by definition identical code is stored in an identical AST. There is only one way to construct an AST for any given Java source code. This allows a survey of the source without having to worry about particular coding styles or formatting. Also, the hierarchical tree structure of the AST allows for easy, predictable traversal.

One method of traversing an AST is use of the Visitor pattern.⁵⁶ The Visitor pattern is a depth-first system of traversing a tree. In the Visitor pattern there are two parts, the tree that it is visiting and a class that “visits” the tree, this being the visitor. All of the nodes of a tree all implement a common interface that requires a method that

⁵⁴ Deryck Brown. Programming Language Processors in Java. [15]

⁵⁵ Skrien. Intro to OOD [12]

⁵⁶ Erich Gamma. Design Patterns [14]

“accepts” the visitor. The first thing the `accept` method does is to call the designated method particular to that type of node in the passed visitor class. In the visitor class there is a separate `visit` method for all the types of nodes that implement the visitor interface. In these `visit` methods, the code specific to that visitor and that type of node is executed including calls to the `accept` methods of the children of that node, thus visiting the entire tree. By using this system, the type of node that is being visited can take care of calling the appropriate function within the visitor. By using this system the traversal of a tree can occur and all code that needs to be executed during the traversal is encapsulated in a single class. This class can perform whatever function that it is designed to do without having to inject code specialized to that function in the AST. The Visitor can perform its function without having to modify the elements it is traversing. The basic structure of the AST does not change; the same types of nodes are used to represent the different parts of the code. Therefore a visitor is a useful tool, in that the functions used to examine the structure of the AST are encapsulated and interchangeable. It is this property of the Visitor pattern that makes it of value in the implementation of JeSS. It is capable of traversing the code structure of a source file in a hierarchical way with interchangeable classes each performing an independent function.

There are also drawbacks to using the Visitor pattern in conjunction with an AST, in that there is a lot of overhead in creating an AST. The structure of an AST is very complex and depends upon the proper parsing of the source code and so the program responsible for the AST must be extensively checked and verified. To create a program that generates an AST is a very serious undertaking all in itself. Rather than trying to do this from scratch it is preferable to use tools to do this automatically. Once the

construction of the AST is complete it is necessary to have an AST that uses the Visitor pattern. This includes having the “accept” method in each of the nodes that makes up the tree as well as a visitor class that has all of the “visit” methods. So the tool used to generate the AST must have some sort of support for the Visitor pattern. The last concern is the structure and documentation of the classes that make up the AST. This concern rises out of the need for the simple creation of visitor classes. It is necessary for the nodes of the AST to have appropriate methods that can be used to discover the properties associated with the node. If this is an undocumented or difficult process then it is much harder task. It is these concerns that focused the implementation choices during the development of the JeSS source code analyzer.

The first tool that was considered was JavaCC.⁵⁷ This tool is a open source project that takes a grammar for a language and automatically generates a parser for it. This can then be used in conjunction with the tool JJTree, which is included in the JavaCC distribution. Together these tools can accept a grammar as input, generate a program that will parse that grammar (JavaCC) and then create a parse tree for it (JJTree). The tools include built-in support for the Visitor pattern. They inject the appropriate `accept` methods into the nodes and produce a skeleton visitor class that include all the needed `visit` methods. However, this package has its shortcomings. First of all the tree that is produced is a concrete parse tree instead of an AST. This means that the nodes that are included on the tree are not all needed when analyzing the source code. For example, a variable name could be stored as a field in the variable declaration node in an AST but in a parse tree the name would be stored in its own

⁵⁷ Java.net. JavaCC Home [29]

branch consisting of the super types of name that the parser uses to define the name (Qualified name, simple name, etc.). In addition to this awkward tree the node classes that are generated by JavaCC are not particularly user friendly. A program generates these classes and as a result they completely lacked documentation. They also lacked methods that could be easily used to determine a node's properties. The use of JavaCC would require writing a program that would prune the parse tree to generate an abstract syntax tree. It also would require extensive work with the generated nodes and visitors to make them user-friendly and easy to understand. As a result, JeSS uses the `org.eclipse.jdt.core.dom` package to generate the AST used in source code analysis.

The `org.eclipse.jdt.core.dom` package is built into the Eclipse IDE distribution. It is used in the eclipse runtime compiler. This package supplies several very attractive features. The generation of the AST is done with a few simple lines of code using the `ASTParser` class found in the package. The generated AST has built in visitor support and included in the package is the `ASTVisitor` class, which is intended to be the superclass for any implemented visitors. Additionally all of the generated classes are well documented in the users manual supplied with Eclipse. This solves all of the problems that arose with the use of the JavaCC tool. As a result this package is what is used in JeSS.

4.3 User Interface

The user interface of JeSS can be split into two parts, the user input and the feedback of the program.

The user input to JeSS should consist of a Java element to be scanned. Ideally the user should be able to select as the element a class, package, or project. This presents the problem of creating a user interface that allows easy selection of elements, but properly restricts them to appropriate types. The feedback that JeSS provides needed to have three major features to be of real use to the user: error-specific messages, dynamic error linking, and a means for correcting errors. An error found by the source code analysis should be shown in a summary panel. This panel should include specific information detailing the type of error and where it occurred. The panel should also dynamically link the summary to the section of the source code corresponding to the error. The source code should be displayed in an editor that allowed for the correction of the error. This editor should, ideally, be a Java editor so that revisions can be made easily.

The first package that was considered for use in creating the UI was the Swing package distributed with Java. This package provides numerous classes for the creation of GUIs. This package was considered because of its ease of use, reliability across platforms and extensive code base. What was soon found is that Swing does not provide tools for what was needed in the JeSS UI. While the Swing package includes very basic tools, it would have required extensive reworking to provide the specific behavior that was desired for JeSS. For example, creating a file chooser that can only select a “.java” file is simple, but creating one that can also choose a directory that only contains “.java” files is not. Also, creating text editors is not a challenging task, but to have dynamic links to sources in the text requires extensive additional code. As this work was not the focus of the JeSS project, other options were explored.

Just as the solution to the problems with the source code analyzer lay in Eclipse, the solution for the UI did as well. During standard use, the Eclipse platform performs all of the functions desired for JeSS. There is built in-error reporting, dynamic error highlighting, and Java editors for the standard use of Eclipse for Java programming. Further investigation revealed that all of these features could be tapped into using the built in plug-in extension points. This realization caused a major change in the project goals for JeSS. It was apparent that the development of JeSS as a standalone application would require the recreation of many features already built into Eclipse. With these considerations in mind the decision was made to develop JeSS solely as a plug-in for Eclipse.

4.4 The Eclipse IDE

The Eclipse IDE is an open source project with the goal of providing “a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools.”⁵⁸ The platform is designed to be fully extensible. The IDE is set up so that its functionality can be dynamically reconfigured using plug-in modules. From the basic functions of the IDE to any number of highly specialized plug-ins, the platform treats them the same. The plug-ins are dynamically discovered and loaded on startup. The default Eclipse package includes a basic set of plug-ins in the Software Development Kit (SDK). This set provides the basic functionality of the platform including java development, and plug-in development. This system allows the seamless integration of “aftermarket” tools. The same systems in which the built-in features interact with each

⁵⁸ Eclipse Foundation. Eclipse.org [33]

other are available to the developers of plug-ins. This basic set-up is what makes Eclipse an attractive IDE to develop plug-ins for. In addition, its widespread use in the US and abroad in both educational and commercial setting make it ideal for JeSS.⁵⁹

⁵⁹ Eclipse Foundation. Eclipse.org [33]

5. JeSS Details

5.1 Goals of JeSS

The chances that a coding tool will be used are greatly diminished if the tool is hard to use. With this in mind JeSS was designed to integrate tightly into the secure code development process. To seamlessly integrate, JeSS was designed in such a way that its use would not be inconvenient or require extensive preparatory work. In addition it was designed to contain a few key features that would promote user acceptance. These requirements were laid out as the design goals for JeSS. JeSS needed to be robust, customizable, extensible, and include dynamic error reporting. Detailed error messages needed to be displayed if a user tries to use JeSS inappropriately. The scans that JeSS performed needed to be customizable. If a bug of a particular type needed to be found the user would not be distracted by error messages referring to other security bugs. The user needed to have a detailed error report explaining each security hole. In addition the error needed to be highlighted in the code. These two parts of the feedback need to be linked so that the possibility of the user confusing one error for another does not come up. Finally the security audit that JeSS performs needs to be able to handle security concerns that may be identified in the future. That is, the end user should be able to scan for security holes that they identify. The functionality of JeSS should not be limited by the knowledge of the original author. These goals guided JeSS' design.

5.2 Layout of the Eclipse IDE

Eclipse consists of a main window that is separated into individual views, and an overhead menu bar. Within the main window the area is separated into two major parts, there is the taskbar and perspective selector and then the individual view windows. The layout within this main window is determined by the “perspective” that is currently in use. The perspective determines what individual views are present and how they are laid out with respect to one another. A general example of this is the Java browsing perspective.

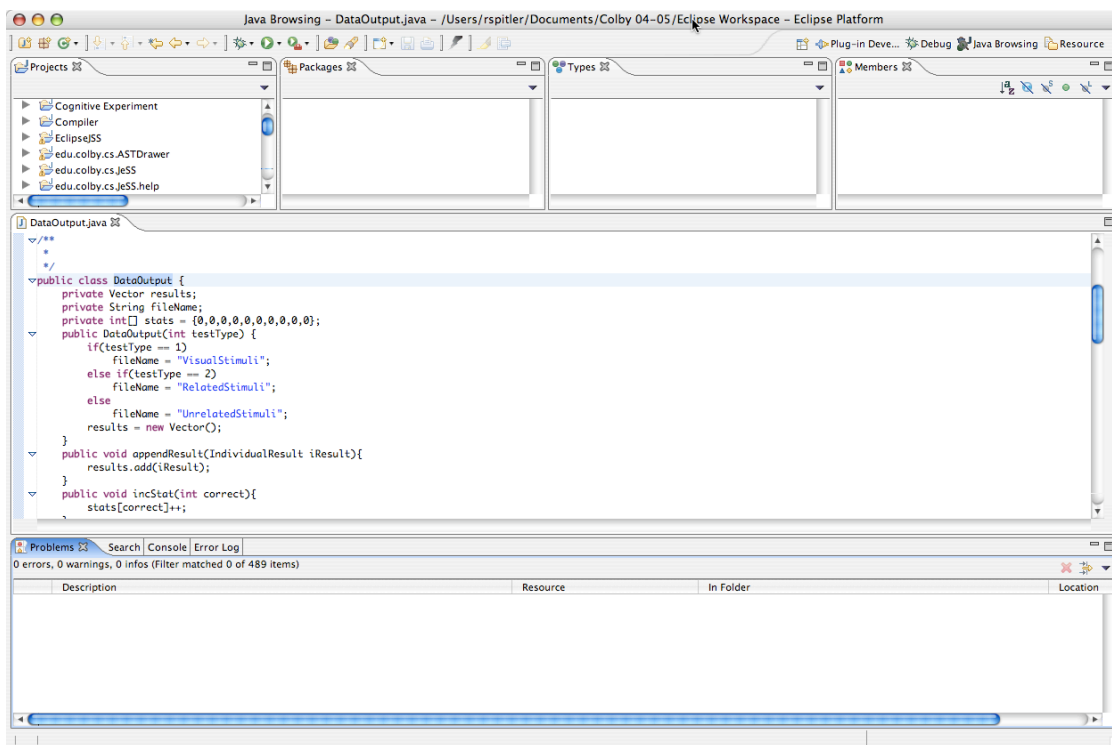


Figure 1: Eclipse showing the Java Browsing Perspective

In this perspective, in the top portion of the main window, there is a view for the available projects, one for the available packages within the selected project and then a

view for the classes within the selected package, and finally a view for the methods of the selected class. In the middle of the main window there is a space in which Java editors are stacked when a class is being edited. At the bottom of the window there are a number of stacked views, the notable ones being a console output, an error log, a problems task list, and a TODO task list. The Java browsing perspective is a typical Eclipse perspective. Other perspectives follow a similar layout, but tailored to accommodate the task of the perspective.

5.3 User Interaction with JeSS

As discussed earlier JeSS' user interface consists of two main parts: the input and the feedback that comes as a result of the input.

In JeSS the user input can be separated into two parts: setting the parameters for the scan and selecting the elements to be scanned. The scan in JeSS is fully customizable by the user. The overhead menu bar contains a menu dedicated to JeSS. In the JeSS menu there is a shortcut to the JeSS preferences (these preferences can also be accessed through the standard path in Eclipse).

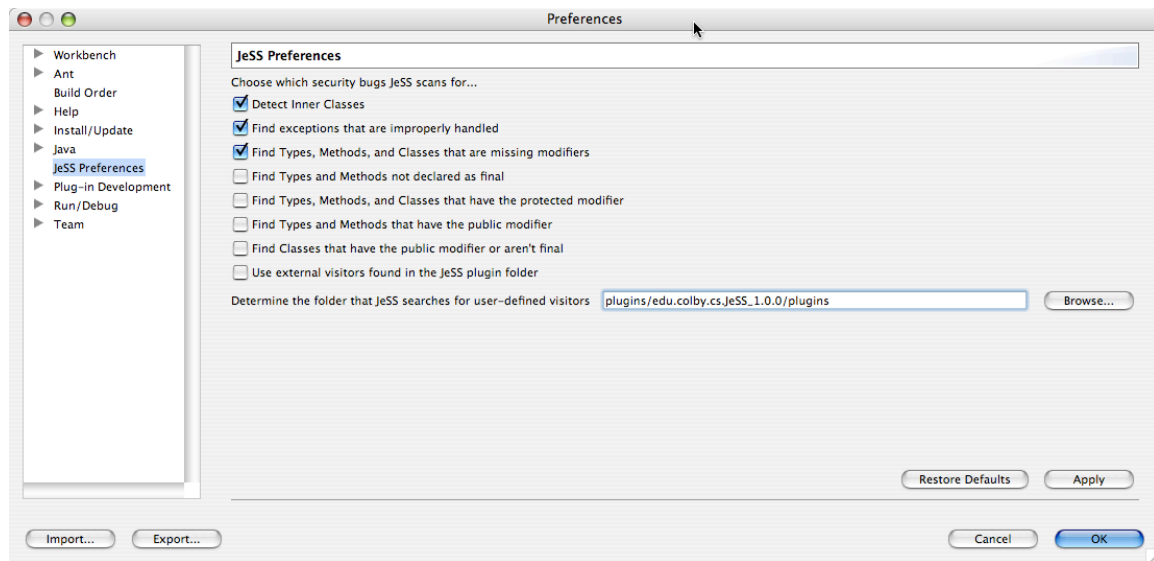


Figure 2: JeSS Preferences

In the JeSS preference menu the user can use a boolean checklist to select the types of scans to be performed. The user can also determine whether they would like JeSS to use their own user-defined visitors in the security audit (there is more information on this in the following section). JeSS searches for the user-defined visitors in the “plugins” directory in the JeSS folder by default. This location can be changed in the preferences dialog. Once the type of scan is set then the user needs to set the input for the scan.

In JeSS the items selected in the views present in the current perspective determine the input of the scan. JeSS is not dependent on the perspective that is currently being displayed in Eclipse. The scan can be run so long as there are views that show a Java element. A Java element consists of a class, package, or project. Depending on the view, these elements can possibly be displayed as a folder representing the underlying file structure of a project or package. Also a class can be displayed as a Java file. JeSS can scan any selection of these types; it doesn't matter if the current view is displaying the element as the underlying file. Every view that displays a Java element can be used

to select the element for the scan. When selecting elements for a scan it is possible to select more than one at a time. For example it is possible to scan all of the projects currently in the workspace by selecting all of these elements in the “Projects” view. If an element of a type that cannot be scanned is selected, then an error message will be displayed that indicates the problem element and will also suggest an appropriate type. Once the user selects all files that are to be scanned they simply need to run the scan using either the JeSS taskbar action or the “Scan Current Selection” option in the JeSS menu (see Figure 3). At this point the source code analyzer is run and then the feedback is displayed.

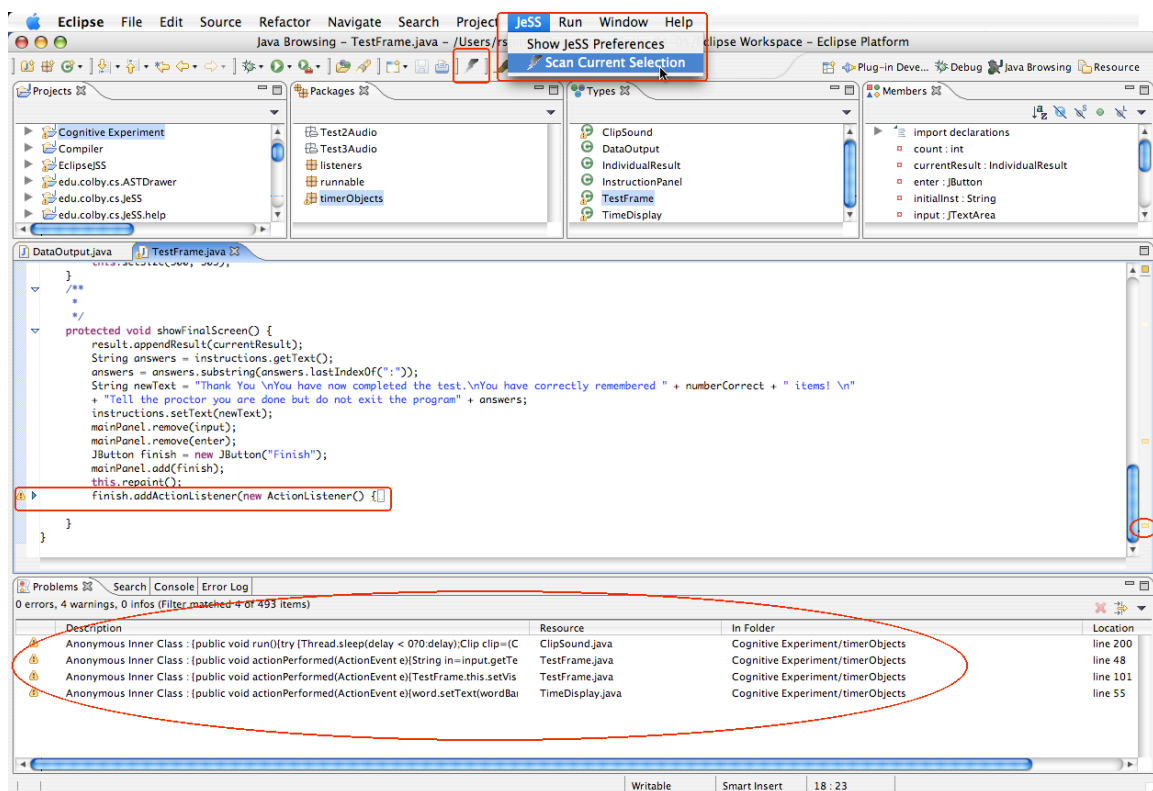


Figure 3: JeSS User Interaction

JeSS provides feedback in three major ways, consisting of a summary in the problems view, highlighted lines in the java editor, and floating icons over the class files.

JeSS also provides the user with rudimentary automated corrections. After the JeSS scan is run, a dialog box appears displaying a short message indicating the number of problems found in all of the classes that were scanned. The “problems” view is also brought to the front of the perspective. In this problems view there is a summary of all the security bugs found by JeSS. The summary includes the files that the bugs were found in, the types of errors, and the line numbers where the bugs can be found in the files. By double clicking any of these summaries the section of the code where the bug is found is automatically opened and highlighted in a java editor. In addition to accessing the bugs in this manner there is also an indicator displayed on the class file. If a security bug is found within the file, then an icon floats over the display of the file. If these files are opened for editing, the security errors are underlined with a squiggly pattern and also marked along the right edge of the editor. Directly to the left of the underlined section of code is an icon that can be right clicked. When the icon is right clicked a list of automated resolutions appears. The current implementation allows the user to ignore the one bug, ignore all bugs in the file, or to ignore all bugs in the project.

5.4 Source Code Auditor

The responsibilities of the source code auditor starts when the user starts the scan. The auditor determines the type of the elements to be scanned, extracts any nested elements (such as a class from a package), produces an AST, determines the scans to be performed, passes all corresponding visitors to the AST, and then reports the results of the scan. In true object oriented style all of these tasks are a performed by a separate class that could be reused, or replaced with minimal refactoring.

To determine the type of element that is to be scanned JeSS uses the JeSSClearingHouse class. The JeSSClearingHouse class has a very simple responsibility, to determine the type of the element, and then pass it along to the SecurityScanner class for nested element extraction and AST generation. The JeSSClearinghouse class contains a series of tests to determine that the selected element is of a Java type. Once this is determined, it identifies the particular type (whether it is a project, package, or class or folder or file representing a project, package, or class) and calls the appropriate method in the SecurityScanner class for the given type.

The SecurityScanner class has two responsibilities: extracting the nested java elements and producing the AST for each base compilation unit (“.java” file). There are three entry points in the SecurityScanner class: scanProject, scanPackage, and scanCompilationUnit. These methods allow the JeSSClearingHouse class to pass any type of Java element along without having to scan all other associated elements. For example one class can be scanned without having to scan all classes in the package, or project. The three main methods in SecurityScanner are arranged in a hierarchical way, As elements are extracted the next method is called. The scanProject method extracts the packages from the project and then iterates through them and calls the scanPackage method on each one. The scanPackage method does essentially the same thing, in that it extracts all the compilation units from a package and then calls scanCompilationUnit on each one. The scanCompilationUnit method produces an AST and then passes it along to the VisitorManager class.

The VisitorManager class manages all of the visitors that perform the actual security audit. This class performs three main functions; it sets up the nature of the scan

according to the current preferences, it takes an AST and performs the scan on it, and it is the class responsible for all of the manipulation of the markers used to indicate the results of the scan. A new instance of this class is generated every time a scan is performed. This is to ensure that the current values set in the JeSS preferences reflect the scan that is performed. In the constructor of this class the preferences are loaded. The values of these preferences then determine what visitors to initialize. It is at this point that JeSS loads user-defined visitors if it is so instructed. Once the visitors have been initialized the scan can commence. The AST is passed to the VisitorManager through a method called “scan.” This method takes the collection of Visitors and passes each of them to the AST. All of these visitors are subclasses of JeSSVisitor. This is for support of user-defined classes (to be detailed in the following section). The JeSSVisitor class contains methods for reporting the errors found. These methods in turn call the reportProblem method in VisitorManager. The actual creation of the problem marker is done in the VisitorManager class, where the particular error is associated with the file that is the root of the AST. Once the VisitorManager has completed the scan of the AST it returns the number of problems that have been found and the job of the source code analyzer is complete.

5.5 Current JeSS Scans

JeSS currently has limited scanning capabilities. The utility of these scans is in reference to the corresponding security hole as detailed in Chapter 3. JeSS is capable of determining the public interface of a project. This scan can be performed at a few different levels. It is possible to find all public methods, classes, and fields. It is also

possible to restrict this scan to solely methods and fields. This can also be done to determine the classes, methods and fields that rely on the package scope. JeSS can identify all methods, fields and classes that are not declared final. The use of inner classes can be automatically detected and exceptions that are improperly handled can also be found. Security holes that are not currently scanned for by JeSS include relying upon initialization, execution of JNI code, proper use of the Cloneable and Serializable interfaces, properly signed code, and encrypted XML generation.

5.6 Extending the JeSS Scans

JeSS supports the use of user-defined visitors in the security audit. After meeting a few structural requirements it is possible for a user to define their own visitor and use it in the JeSS scan. This feature can be enabled in the JeSS preferences.

The implementation of user-defined visitors has been simplified as much as possible. To create such a visitor a user must identify a security hole and the corresponding AST structure for the hole. They must then create the visitor class that can automatically identify this structure. This visitor must be a subclass of JeSSVisitor. JeSSVisitor is, itself, a subclass of the ASTVisitor class found in `org.eclipse.jdt.core.dom`. The ASTVisitor class is the type required by the AST's used in JeSS. Past this, the JeSSVisitor provides a few helper methods for the implemented subclasses. There is reportProblem method that takes a node and an error string. This method formats the error and passes it to the VisitorManager for error reporting. This ensures that the problems found by the user-defined visitors are reported in the same way as other JeSS problems. It also removes from the user the responsibility of creating the

problem marker. There are also a few helper methods that parse the name of the class or method from a corresponding node. In addition to being a subclass of `JeSSVisitor` the visitor must have a constructor that takes the type `VisitorManager` as a parameter. This constructor must then, in turn, call the constructor in the super class taking the same parameter. A more detailed explanation of creating a `JeSSVisitor` can be found in the `JeSS` users manual.⁶⁰

In the `JeSS` preferences dialog the user can enable the use of external visitors. In this dialog the user also has to specify the location of the “.class” file of the visitor. Once this has been done `JeSS` dynamically loads the visitors when the `VisitorManager` class is initialized. To dynamically load these visitors `VisitorManager` creates an instance of the `JeSS` class `PluginLoader`. `PluginLoader` uses the directory path found in the preferences to locate visitor class files. It loads all “.class” files found in this directory. This is done through a private delegate of the local class loader. Then through reflection, the `PluginLoader` ensures that these files are a subtype of `JeSSVisitor` and have a constructor that takes the type `VisitorManager` as a parameter. Once the visitors have been confirmed to have the proper type, reflection is used again to instantiate the classes. The instantiated classes are then returned to the `VisitorManager` for use in the security audit.

5.7 Extending the JeSS Plug-in

Throughout the design and implementation of `JeSS` extensibility has been a consideration. As a result there are no constructs that limit the future uses of `JeSS`. To extend the current scanning functionality of `JeSS` it is as simple as creating a new visitor

⁶⁰ Appendix A

(as explained in the previous section). To include this new visitor as a built-in scan requires modification of two classes; the `VisitorManager` and the `JeSSPreferences`. Changing the user interface only requires changed the points in which JeSS plugs into the Eclipse platform. To create a separate view in which JeSS reports its problems would only require creating the new view and changing the way `VisitorManager` creates the markers. The future uses of JeSS are only limited by the imagination of its users. The highly customizable nature of the Eclipse IDE allows JeSS to be incorporated in whatever way is deemed necessary. In addition to this, the `org.eclipse.jdt.core.dom` package is being updated continuously to accommodate new versions of Java.⁶¹ The AST's generated will reflect the newest version of Java supported by the Eclipse platform. As this project was developed as open-source, the source code will be distributed along with the plug-in. This will allow the end users to customize their versions of the plug-in and improve upon its source.

The mobility of JeSS is a harder issue to address. JeSS' user interface is tightly integrated with the Eclipse platform. To migrate this plug-in to another IDE would require the refactoring of the classes dealing with Eclipse specific elements. However, the basic function of all these classes would remain the same in another platform. So the refactoring would likely just be the replacement of Eclipse specific packages with the corresponding foreign packages. The backend of JeSS is also dependent on Eclipse packages, however it is not as tightly integrated. The AST and the visitors are derived from the `org.eclipse.jdt.core.dom` package. It would not be a difficult undertaking to refactor the backend so that it uses another tool for AST generation. As with the user

⁶¹ Eclipse Foundation. Eclipse.org [33]

interface the Eclipse DOM specific classes would have to be replaced with the corresponding foreign classes. This is actually a simple process, as it was done when the transition from JavaCC to org.eclipse.jdt.core.dom first took place. All other classes dealing with the actual source auditing would be reusable. The VisitorManager and PluginLoader are not largely dependant on Eclipse.

6. Results of the JeSS Project

6.1 Goals Met

The JeSS project was largely a success. The current implementation of the JeSS scanner has met all but one of the original goals of the project. It is a fully customizable scanner. The user can use the JeSS preferences to determine the types of scan that they need. It is possible to perform scans due to programmer or design oversights, such as missing modifiers and catch blocks without any actions. JeSS can be used as a final auditing tool to discover the entire user interface (all public variables, methods and classes), as well as protected or non-final ones. It is also capable of highlighting problematic structures in the code, such as anonymous inner classes.

However, the functionality of JeSS is not what was originally envisioned, as all of the guidelines set out in Gary McGraw's paper are not scanned for. The missing functionality includes relying upon initialization, properly signing code, the proper use of the Cloneable and Serializable interfaces. The additional security holes that were identified but not yet implemented are the encryption of XML generation, and the execution of JNI code. This shortcoming is somewhat accommodated by the support of external security auditors. The creation of external security auditors is far easier than it was originally thought to be. Through subclassing the user does not have to know any eclipse specific knowledge, they can simply find their problem and then report with the built-in method reportProblem. Knowledge of the creation of the problem markers is not needed, nor is any other low level details of the JeSS implementation. JeSS is truly versatile as a result of this feature. There are no real restrictions on the external visitors. The end user can define a visitor that performs any sort of function on the source code

not just security auditing. JeSS provides an automated tool for deploying the visitor on a large scale. Through thorough testing and extensive error catching code, JeSS is capable of dealing with any input that the user provides. This input is screened and the user is notified through detailed error messages on whether it is an acceptable type. This makes JeSS a robust tool; it is capable of performing under any condition that eclipse can produce.

6.2 Deployment of JeSS

JeSS will be distributed on a maintained website dedicated to the tool. It is deployed as a plug-in for Eclipse 3.0. The plug-in includes an extensive users manual,⁶² the source code⁶³, and a “readme” file⁶⁴ that details the first time use of the plug-in. The JavaDoc for the plug-in is also included.⁶⁵ The current plan is for the website to provide periodic updates as well as a system for users to post the code for their security visitors.

⁶² Appendix A

⁶³ Included on CD

⁶⁴ Appendix B

⁶⁵ Appendix C

6.3 Future Work on JeSS

The first step is to make JeSS a fully functional security scanner. This would include implementing visitors for all of the security holes that were researched. These visitors would then be built into JeSS and the preferences would include the options of performing those scans. Another feature that would greatly add to the utility of JeSS is a separate view for the reporting of JeSS problems. With some types of scans, such as the public interface discovery, the scanner can flood the problems view. To separate out these scans would allow the user to use JeSS while still being able to refer to any other problems that are displayed in that view. The automated resolutions also need to be fully implemented. Currently there is only support for removing the markers. There needs to be suggested resolutions for the particular security holes. Creating a system in which users can provide resolutions for their own security visitors could further extend this aspect of JeSS. The class supplying the resolution would be incorporated into the visitor that the user provides. With the current implementation of JeSS this work can be easily done. There is no major hurdle to incorporating any these features other than the commitment of time to make them wo

7. References

Java and Software Security

Books and Articles:

1. Chris Hawblitzel, C.-C. C., Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken (1998). Implementing Multiple Protection Domains in Java. USENIX Annual Technical Conference, New Orleans.
An example of extending Java's sandbox model to enhance the security of applets
2. Gary McGraw, E. F. Twelve rules for developing more secure Java code. 2004.
The basis of my project, a brief article detailing the basic java security problems
<http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html>
3. Gary McGraw, E. F. (1999). Securing Java: Getting Down to Business with Mobile Code, Wiley.
An extensive look at all aspects of Java security, from both the users and programmers P.O.V.
<http://www.securingsjava.com/>
4. Gutschmidt, T. Securing Java Code: Part 1. 2004.
Outline of a corporate policy to promote secure java programming
<http://www.developer.com/java/article.php/741921>
5. J. Steven Fritzinger, M. M. (1996). Java Security, Sun Microsystems, Inc.
The basic outline of the java security model
6. John Viega, G. M., Tom Mutdosch, Edward Felten (2000). "Statically Scanning Java Code: Finding Security Vulnerabilities." IEEE Software: 68-74.
An expanded version of [2] with information on static analysis
7. Kalinovsky, A. (2004). Covert Java, Sams.
A detailed look at the system of decompiling Java byte code and other methods to compromise java programs
8. McGraw, G. (1998). Priviledged code in Java.
Explanation of the Priviledged code API and its use in Java programming
<http://www.developer.com/java/other/article.php/604131>
9. McGraw, G. (2004). "Software Security." IEEE Security & Privacy.
A look at the design process for developing secure software

10. Nolan, G. (2004). *Decompiling Java*, APress.
A good source for information on decompiling Java byte code and other methods of compromising Java programs
11. S. Doyon, M. D. (2000). "On object initialization in the Java bytecode." *Computer Communications*(23): 1594-1605.
A look at the process of object initialization from a low level perspective containing possible problematic constructor code
12. Skrien, D. *Intro to OOD. Work In Progress*
An introduction to programming with an Object Oriented approach with a detailed look at many OO design principles.
13. John Viega, J.T. Bloch, G.M., Tadayoshi Kohno (2000) "its4: A Static Vulnerability Scanner for C and C++." ACSAC Technical Conference.
A look at static scanning of C and C++ code. An overview of security problems and implementation of a scanner
14. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. (1995) *Design Patterns*, Addison Wesley.
The definitive guide to design patterns including a detailed description of the Visitor pattern.
15. Deryck Brown, David Watt. (2000) *Programming Language Processors in Java*, Prentice Hall.
A look at compiler construction in Java, contains information on production of AST

Websites:

16. Sun Microsystems. *Security and the Java Platform*. 2004.
The up-to-date outline of Java security procedures
<http://java.sun.com/security/index.jsp>
17. Sun Microsystems. *Security in Object Serialization*. 2004.
Security concerns when using the serialization API
<http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/security.doc3.html>
18. Sun Microsystems. *Serialization Specification*. 2004.
The outline of the serialization API
<http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serialTOC.html>
19. Sun Microsystems. *Java Security Architecture 1.5*. 2004.
The security architecture in Java 1.5
<http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-specTOC.fm.html>

20. Sun Microsystems. Applet Security. 2004.
FAQ on applet security and privileges
<http://java.sun.com/sfaq/>
21. Sun Microsystems. Handling Errors with Exceptions. 2004.
A general look at the proper use of exceptions to deal with errors generated in java code
<http://java.sun.com/docs/books/tutorial/essential/exceptions/>
22. Sun Microsystems. Reflection. 2004
Overview of the Reflection API
<http://java.sun.com/j2se/1.3/docs/guide/reflection/>
23. Sun Microsystems. Working with XML. 2004.
A general look at the use of the JAXP XML package in Java
<http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/index.html>
24. Sun Microsystems. Java Architecture for XML Binding (JAXB). 2004.
The overview of the JAXB XML package in Java
<http://java.sun.com/developer/technicalArticles/WebServices/jaxb/index.html>
25. Sun Microsystems. Overview of the JNI. 2004.
A look at use of the Java Native Interface API
<http://java.sun.com/docs/books/tutorial/native1.1/concepts/index.html>
26. Sun Microsystems. Cloning Objects. 2004.
A look at the Cloneable API
<http://java.sun.com/developer/JDCTechTips/2001/tt0306.html>
27. Tech FAQ. How can I find security vulnerabilities in my source code? 2004.
A list of security scanners for C/C++ code
<http://corky.net/2600/computers/source-code-security-vulnerabilities.shtml>
28. Fortify Software. Fortify. 2004.
The home site for the sole Java Security Auditor currently available
<http://www.fortifysoftware.com/products/scal>
29. Java.net. JavaCC Home. 2004
The home site for the parser generator software JavaCC and the tree generator jjTree.
<https://javacc.dev.java.net/>

Eclipse & Plug-in Development

Books and Articles:

30. John Arthorne, C.L. (2004). Official Eclipse 3.0 FAQs, Addison Wesley.
A comprehensive FAQ supplying short answers to questions about many aspects of Eclipse and Eclipse Plug-in development
31. Eric Clayberg, D.R. (2004). Eclipse: Building Commercial-Quality Plug-ins, Addison Wesley.
A step by step guide to producing a fully featured functional plug-in
32. Dejan Glazic, J.M. (2001). Mark My Words, IBM.
A detailed look at implementing markers in eclipse

Websites:

33. Eclipse Foundation. Eclipse. 2005.
The eclipse website. Source code, JDE and developer support.
www.eclipse.org
34. Eclipse Plug-in Central Alliance. Eclipse Plug-in Central. 2005.
A central database of eclipse plug-ins giving short descriptions and links to plug-in homepages
<http://www.eclipseplugincentral.com/>
35. Eclipse Wiki. 2005.
A limited resource for information on various aspects of eclipse development
<http://eclipsewiki.editme.com/>

Appendix A – JeSS Users Manual

The users manual is implemented as a simple plug-in that is accessed along with the standard Eclipse help. The table of contents is displayed along with the other help topics found in the Eclipse manual.

Contents:

Introduction	
<i>Getting Started</i>	61
<i>Simple Use</i>	61
<i>Advanced Options</i>	64
Concepts	
Java Security	66
Extending JeSS	
<i>JeSS Basics</i>	66
<i>Implementing a JeSSVisitor</i>	67
<i>JeSSVisitor</i>	69
<i>Example JeSSVisitor</i>	71

Introduction

JeSS is an automated tool to detect static security flaws. Resources will be scanned for code structures that could be used to compromise proprietary code, or to escalate local privileges. The details of these code structures are explained in the Concepts section of this user's guide.

Getting Started

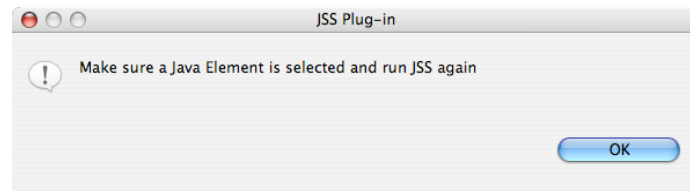
Before you can use JeSS, you will need to start up Eclipse with the JeSS plugin and open the project containing the Java source code you want scanned.
See appendix B for the readme explaining how to do this.

Simple Use

Selecting Objects

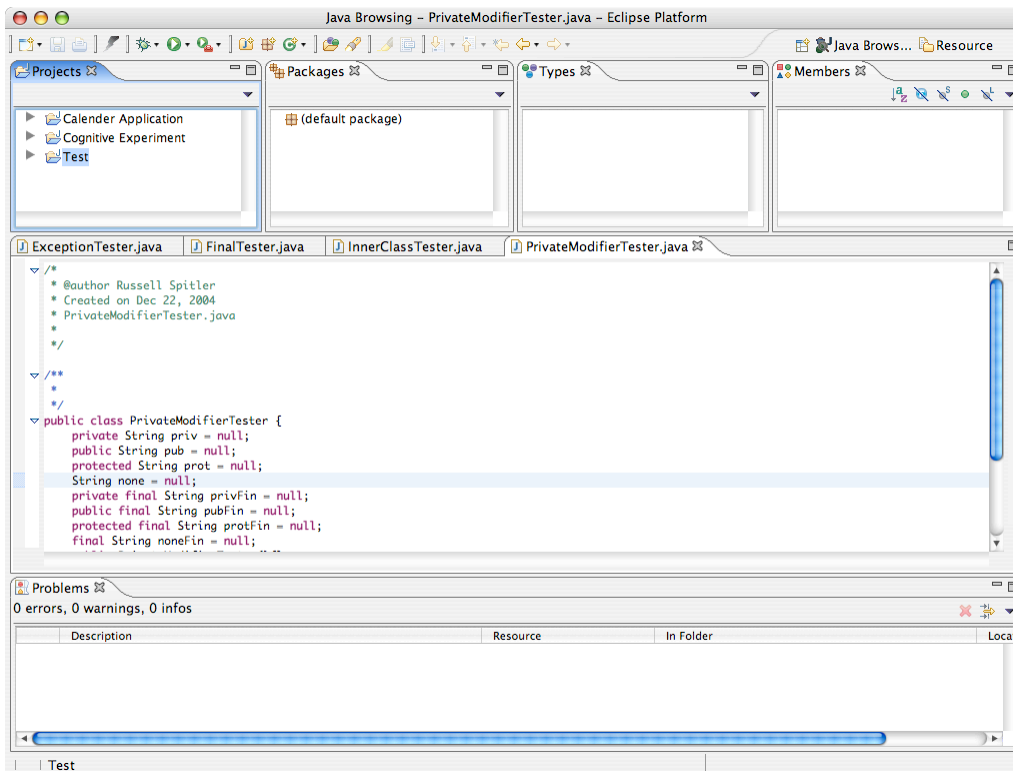
To start a security scan, select one or more Java elements in one of the Eclipse navigator windows. A Java element includes any Java project, package, or “.java” file. In different Eclipse views these are represented as files, class objects, and folders. JeSS works with

items selected in the Navigator view, Package Explorer, and all views in the Java Browsing perspective. If an element that is of a type that cannot be scanned by JeSS an error message will appear when the scan is initiated.




Example Error Message: if an invalid selection is made an error will occur

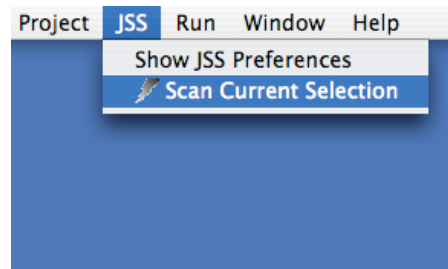
This can be confusing in Eclipse as elements can be highlighted without having the actual focus of the workspace. Make sure that the element that is to be scanned is both highlighted and has the focus of the workspace.



Example Selection: in this view the Java Project “Test” is selected

Performing a Scan

The next step is to use either the JeSS scan button () in the task bar at the top of the workspace or to select the “Scan Current Selection” item from the JeSS menu.

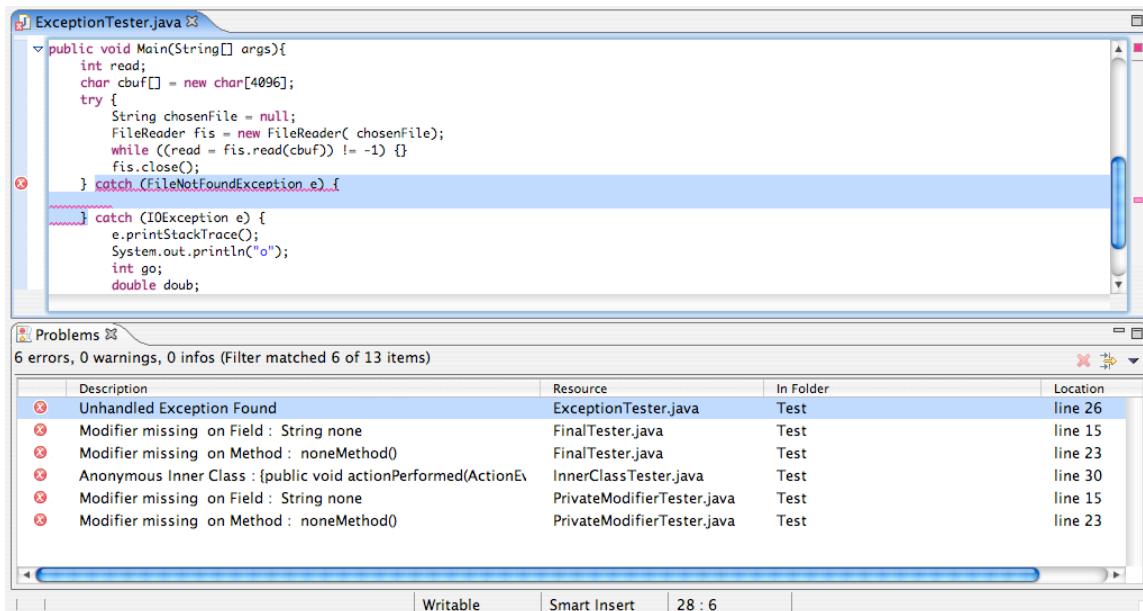


Example Menu: Use the “Scan Current Selection” action

This starts the JeSS scan. The progress bar at the bottom of the workspace will show the progress of the JeSS scan. If more than one project is being scanned then this may take a few seconds.

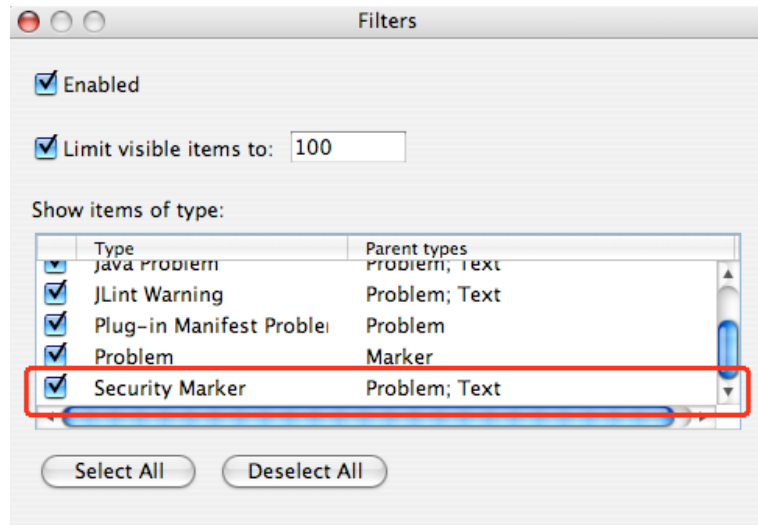
Dealing with the Results

After the JeSS scan is completed a dialog will appear saying how many problems were found during the scan. In addition to this the “Problems” view will be brought to the front of the workspace. In this view, the problems discovered by the JeSS scan will be displayed.



Example problem: Unhandled Exception

IMPORTANT: if JeSS has found problems but they do not show up in the “Problems” dialog then make sure that the filter on the problems view is not excluding JeSS problems from the view



Example Problem View Filter: make sure the “Security Marker” is checked

The problems found by JeSS and listed in the Problems view can be treated like any other problem in that view. For example, to jump to a problematic section of your code, simply double click on the problem in the view and an appropriate editor will appear with the code highlighted. JeSS does not supply any true resolutions to these problems, but you can choose to ignore the particular warning, ignore all warnings in that file, or ignore all warnings on the project. A better solution would be to modify your code to eliminate these problematic sections as outlined in the [Concepts](#) section of this users guide.

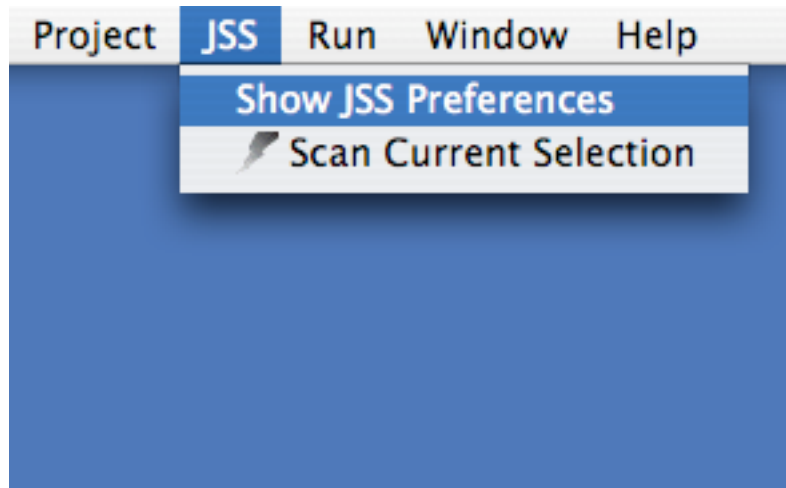
Advanced Options

Scan Types

With JeSS you can customize the set of security bugs that will be scanned for. As default JeSS will only scan for Anonymous Inner Classes, Unhandled Exceptions, and Missing Modifiers (details of these security bugs can be found in [Concepts](#) section). JeSS can also determine the methods, fields and classes of a project that are declared public. A similar scan can also be used to find all protected methods, and to find all classes, methods and fields that are not declared final. These types of scans should be done individually as they typically produce a large number of results. The utility of these types of scans is described in detail in the [Concepts](#) section.

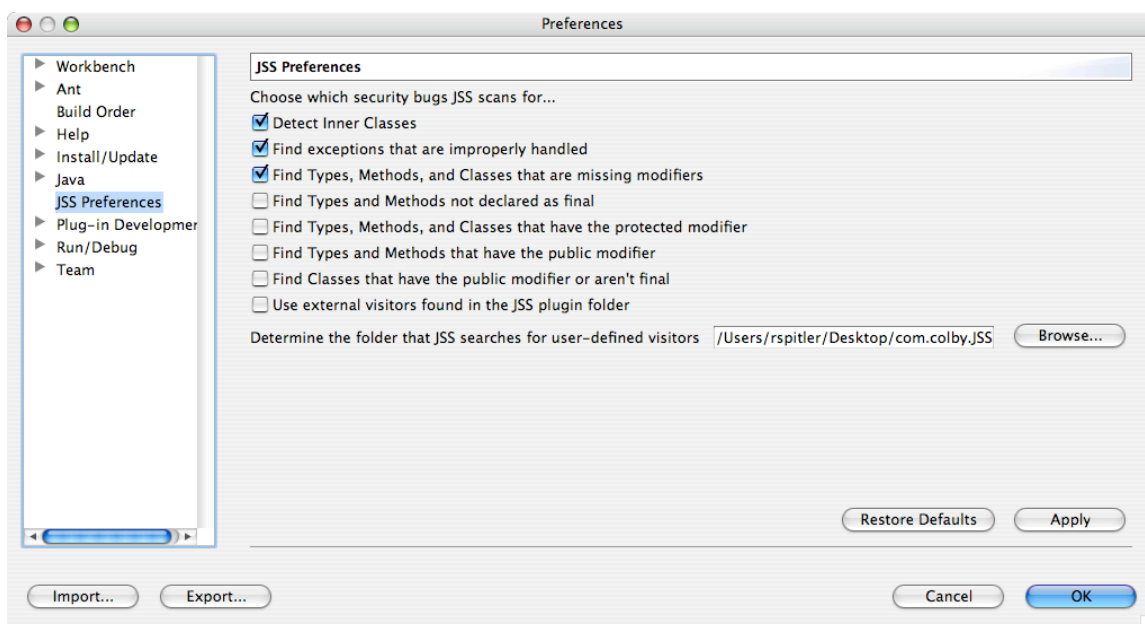
Selecting Scan Types

It is a simple process to customize the scan that JeSS performs. This can be done on the JeSS Preference page. To access the JeSS Preference you can either select the “Show JeSS Preferences” from the JeSS menu or access them through “Window-->Preferences” and then select “JeSS Preferences” from the menu on the left hand side of the dialog.



Select “Show JeSS Preferences”

This dialog shows a simple list of available scanners with a Boolean checkbox next to them. To enable or disable a particular type of scan simply select or deselect the checkbox that corresponds to the desired scanner. The last two elements on this preference page are to be used to extend the functionality of JeSS.



JeSS Preferences: use the checkboxes to customize the security scan

Using Your Own Scanners

Greater detail on this subject can be found in the [Extending JeSS](#) section of this users guide.

Concepts of JeSS

Java Security

Java is a language that has been designed from the beginning with security in mind. [5] It is implemented with the Sandbox model in order to limit the privileges of running code. The language has been designed to prevent major security errors such as buffer overflows. Numerous precautions have been made to ensure that java is a truly mobile secure language. A detailed description of the java security model can be found at [13] & [16]. With this in mind there are still many steps that a programmer can take in order to further secure their own code. This is done to protect proprietary code and to prevent their application from being used in an attack to escalate privileges. The process of producing secure java code is examined in [3]. This is further reduced to a paper on twelve basic rules for more secure java code [2]. In the following section a more detailed look at these twelve principals as well as a few additional ones is provided. While JeSS does not currently scan for all of these security bugs, external visitors can be implemented to do so.

The 11 security hole explanations, as found in the main paper, are found in this section of the Users manual

Extending JeSS

When JeSS scans a file it first produces an abstract syntax tree (AST). JeSS then searches this AST for problematic code structures using the Visitor pattern. A visitor is passed to the root of the AST and it searches for the signature of security bugs. Each security bug is scanned for using a separate visitor. This structure allows JeSS to be easily extended. To scan for new security bugs simply introduce a new visitor of the appropriate type.

JeSS Basics

AST Production

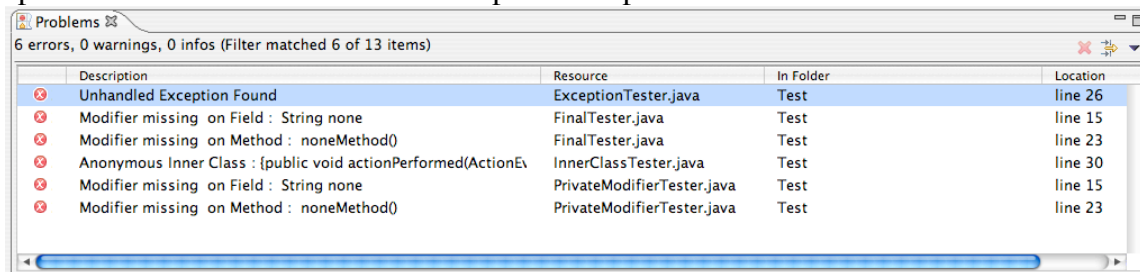
JeSS produces an AST using the built in org.eclipse.jdt.core.dom package. The class ASTParser is used to accept the compilation unit and produce an AST. Manipulation and use of this AST requires the use of the classes in the org.eclipse.jdt.core.dom package. Visitors that are passed to this tree must be of type ASTVisitor.

Visitors and JeSS

The visitors used in JeSS are a sub-type of ASTVisitor. They are of type JeSSVisitor. JeSSVisitor is further sub-classed to create the individual visitors that are used to scan for the security bugs. JeSSVisitor provides three helper methods for the security scans. There are methods to report a problem, to parse a class name out of a node, and to parse a method or field name out of a node. Greater detail on this class can be found in the section JeSSVisitor.

Reporting Problems

In JeSS problems found are displayed in the “Problems” view. These markers are a sub-type of org.eclipse.resources.problemmarker. The standard format for reporting a problem in JeSS is a general message conveying the nature of the security bug and then a specific reference associated with the particular problem.



For example a field that is missing a modifier would be reported as follows

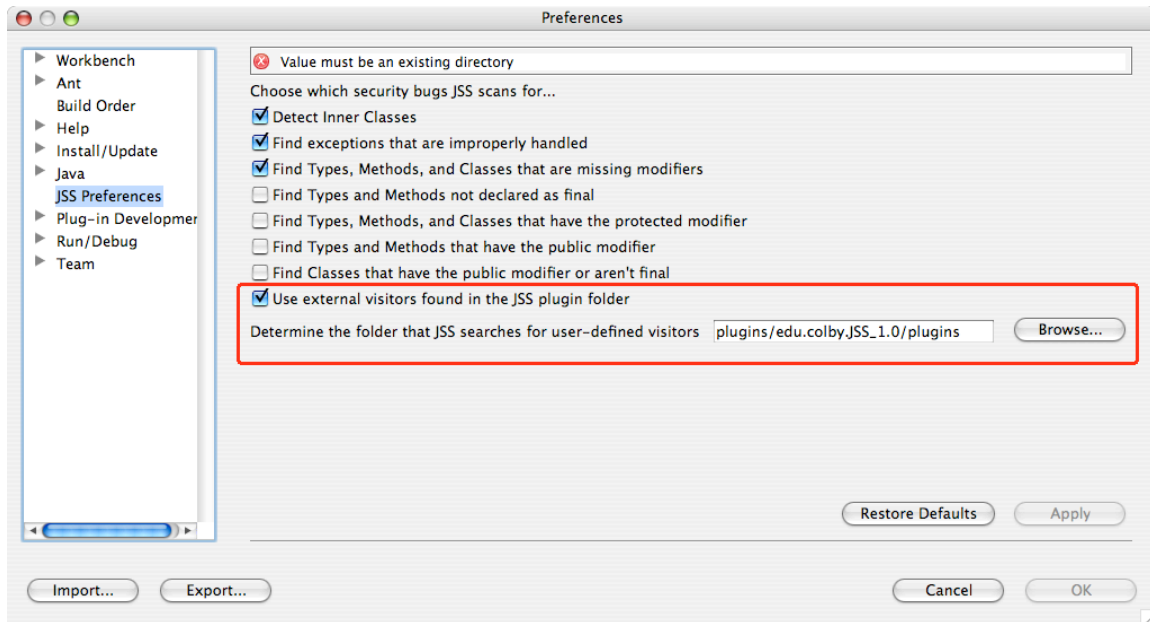
“Modifier missing on Field: String none.”

This convention should be used in all implementations of JeSSVisitor.

Implementing a JeSSVisitor

Plugging into JeSS

It is possible to design and use your own visitors with the JeSS plug-in. In order to do so a visitor of the appropriate form is required. There is a more detailed description of the requirements in the following sections. In order to plug in your visitor you must enable the use of external visitors in the preferences page. To do so open up the JeSS Preferences page, this can be done through the JeSS menu or Window→Preferences→JeSS Preferences. Once in the preference page select the Boolean checkbox corresponding to the “Use external visitors found in the JeSS plug-in folder” option.



JeSS Preferences: Enable the use of external visitors and select the directory where they are to be found

This causes JeSS to search for visitors in the specified directory. The next field allows the user to specify the directory that JeSS searches for the visitors in. The default directory is a folder labeled “plugins” in the local JeSS plug-in directory. For example from the eclipse folder the plugins would be found on the path:

~/eclipse/plugins/edu.colby.JESS_1.0/plugins

If another directory is desired then it can be specified in the Preference page.

Requirements

Once the use of external visitors has been enabled there are a few requirements for an external visitor. JeSS requires the following of an external visitor:

- The visitor must be a “.class” file. JeSS will not compile the visitor for you
- The visitor must extend [JeSSVisitor](#).
- The visitor must have a constructor that takes type `edu.colby.JeSS.scanner.VisitorManager` as a parameter and call the constructor in `JeSSVisitor` that takes this parameter.
- The visitor “.class” file must be in the directory specified in the preference page

Design

In order to uncover security bugs a [JeSSVisitor](#) must be designed to recognize and flag certain code structures that signify the security bug. The signature that the Visitor searches for is specific for each security bug, but the process of reporting this problem is built into the [JeSSVisitor](#) class. Implementing your own [JeSSVisitor](#) can be done very easily. The basic process can be broken down into the following steps:

- Identify a security bug and determine what the signature looks like in the AST
- Create a visitor class that can recognize this signature

- Make the visitor a subclass of [JeSSVisitor](#) and use the `reportProblem()` method to mark the problematic sections of the code

The hardest part of creating your own visitor is the automated recognition of the problematic AST structure. It may be helpful to examine the documentation for the AST nodes that the visitor will traverse. This can be found in the `org.eclipse.jdt.core.dom` API. An [example of a visitor](#) that identifies unhandled exceptions can be found in the following section. However, once your visitor knows how to find the security bug it is a simple matter to report a problem using the `reportProblem()` method.

JeSSVisitor

```
package edu.colby.JeSS.util;
import org.eclipse.jdt.core.dom.*;
import edu.colby.JeSS.scanner.VisitorManager;

/**
 * This is the super class for all Visitors in the JeSS scanner.
 * This is created to allow easy extensibility to the JeSS plugin.
 * Simply create a sub-type of JeSSVisitor to find patterns in an
 * AST and then use the reportProblem() method of JeSSVisitor to
 * create a security marker.
 * @author Russell Spitler
 * Mar 24, 2005
 */
public class JeSSVisitor extends ASTVisitor {

    private VisitorManager vManager;
    //store a reference to the Visitor manager for error reporting
    public JeSSVisitor(VisitorManager vManager){
        this.vManager = vManager;
    }
    /**
     * Used to report a problem and create a security
     * marker for the security bug. This method uses
     * the reportProblem() method of the VisitorManager.
     * @param node - the root of the problem
     * @param errorMessage - the message associated with the error
     */
    public void reportProblem(ASTNode node, String errorMessage){

        Location loc = new Location();

        CompilationUnit compUnit = (CompilationUnit) node.getRoot();
```

```

        loc.setLineNumber(compUnit.lineNumber(node.getStartPosition()));
        loc.setCharEnd(node.getStartPosition()+node.getLength());
        loc.setCharStart(node.getStartPosition());
        loc.setFile(vManager.getResource());

        vManager.reportProblem(errorMessage, loc, true);
    }
    /**
     * This helper method parses a class name from the output of
     * the standard toString() method in the TypeDeclaration
     * AST node. This method relies upon the standard format of
     * TypeDeclaration[class CLASSNAME DECLARATIONS]. The name
     * is converted to user readable form "class CLASSNAME"
     * @param string - toString() from a TypeDeclaration AST node
     * @return the name in user readable form
     */
    protected String parseClassName(String string){

        //start after the first [
        int startIndex = string.indexOf("[")+1;

        //end after the first space following "class "
        int endIndex = string.indexOf(" ", startIndex+7);
        string = string.substring(startIndex, endIndex);
        return string;
    }
    /**
     * This method parses a user readable name from the
     * toString() output of FieldDeclaration and MethodDeclaration.
     * This method relies on the standard format of
     * ***Declaration[***NAME] where *** is either Type or
     * Method
     * @param string - toString() from a MethodDeclaration or Field
     Declaration *          node

     * @return the name in user readable form
     */
    protected String parseStandardName(String string){

        string = string.substring(string.indexOf(" "), (string.length()-1));
        return string;
    }
}

```

Example JeSSVisitor – Unhandled Exception Finder

```

/*
 * @author Russell Spitler
 * Created on Dec 26, 2004
 * ExceptionFinder.java
 *
 */
package edu.colby.JeSS.visitors;

import java.util.List;
import org.eclipse.jdt.core.dom.*;
import edu.colby.JeSS.scanner.*;
import edu.colby.JeSS.util.JESSVisitor;

public class ExceptionFinder extends JeSSVisitor {

    private String errorMessage = "Unhandled Exception Found";

    public ExceptionFinder(VisitorManager vManager){
        super(vManager);
    }

    public boolean visit(CatchClause node) {
        //retrieve the block contained in this catch clause
        Block body = node.getBody();
        //retrieve the statements contained in this block
        List l = body.statements();
        //if there are no statements in the block report error
        if(l.isEmpty()){

            reportProblem(node, errorMessage);
        }
        return true;
    }
}

```

In the Reference section of the Users Manual the References as found in section 7 of the main paper are found. As well as the javaDoc for the project, which is found in Appendix C.

Appendix B – JeSS README

Installing JSS:

Drag the edu.colby.cs.JeSS plugin directory and the edu.colby.cs.JeSS.help directory (found with this file) into the plugin directory found in the Eclipse directory. Start Eclipse. It is possible that you will need to start Eclipse with a clean build in order to register the plugin.

Once Installed:

Make sure that the "Problems" view is not filtering out the JSS markers. This can be done by clicking the filter icon in the upper left hand corner of the Problems view. Then scroll down to the bottom of the list of markers and select the boolean checkbox next to the JeSS Security Marker. More information on this can be found in the JSS User manual, which can be accessed through standard eclipse help (Help--> Help Contents).

Appendix C - JavaDoc

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS

Class JeSSPlugin

```
java.lang.Object
  extended by org.eclipse.core.runtime.Plugin
    extended by org.eclipse.ui.plugin.AbstractUIPlugin
      extended by edu.colby.cs.JeSS.JeSSPlugin
```

All Implemented Interfaces:

org.osgi.framework.BundleActivator

public class **JeSSPlugin**

extends org.eclipse.ui.plugin.AbstractUIPlugin

The main plugin class to be used in the desktop. This class is generated by Eclipse.

Nested Class Summary

Nested classes inherited from class org.eclipse.ui.plugin.AbstractUIPlugin

Field Summary

private static JeSSPlugin	plugin
private java.util.ResourceBundle	resourceBundle

Fields inherited from class org.eclipse.ui.plugin.AbstractUIPlugin

Fields inherited from class org.eclipse.core.runtime.Plugin

PLUGIN_PREFERENCE_SCOPE, PREFERENCES_DEFAULT_OVERRIDE_BASE_NAME,
PREFERENCES_DEFAULT_OVERRIDE_FILE_NAME

Constructor Summary

[JeSSPlugin](#)(org.eclipse.core.runtime.IPluginDescriptor descriptor)

The constructor.

Method Summary

<code>static JeSSPlugin</code>	<code>getDefault()</code> Returns the shared instance.
<code>java.util.ResourceBundle</code>	<code>getResourceBundle()</code> Returns the plugin's resource bundle,
<code>static java.lang.String</code>	<code>getResourceString(java.lang.String key)</code> Returns the string from the plugin's resource bundle, or 'key' if not found.
<code>static org.eclipse.core.resources.IWorkspace</code>	<code>getWorkspace()</code> Returns the workspace instance.

Methods inherited from class org.eclipse.ui.plugin.AbstractUIPlugin

`createImageRegistry, getDialogSettings, getImageRegistry, getPreferenceStore, getWorkbench, imageDescriptorFromPlugin, initializeDefaultPluginPreferences, initializeDefaultPreferences, initializeImageRegistry, loadDialogSettings, loadPreferenceStore, refreshPluginActions, saveDialogSettings, savePreferenceStore, shutdown, start, startup, stop`

Methods inherited from class org.eclipse.core.runtime.Plugin

`find, find, getBundle, getDescriptor, getLog, getPluginPreferences, getStateLocation, internalInitializeDefaultPluginPreferences, isDebugging, openStream, openStream, savePluginPreferences, setDebugging, toString`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Field Detail

plugin

```
private static JeSSPlugin plugin
```

resourceBundle

```
private java.util.ResourceBundle resourceBundle
```

Constructor Detail

JeSSPlugin

```
public JeSSPlugin(org.eclipse.core.runtime.IPluginDescriptor descriptor)
```

The constructor.

Method Detail

getDefault

```
public static JeSSPlugin getDefault()
```

Returns the shared instance.

getWorkspace

```
public static org.eclipse.core.resources.IWorkspace getWorkspace()
```

Returns the workspace instance.

getResourceString

```
public static java.lang.String getResourceString(java.lang.String key)
```

Returns the string from the plugin's resource bundle, or 'key' if not found.

getResourceBundle

```
public java.util.ResourceBundle getResourceBundle()
```

Returns the plugin's resource bundle,

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.visitors

Class ExceptionFinder

java.lang.Object

extended by org.eclipse.jdt.core.dom.ASTVisitor

extended by [edu.colby.cs.JeSS.util.JeSSVisitor](#)

extended by **edu.colby.cs.JeSS.visitors.ExceptionFinder**

public class **ExceptionFinder**

extends [JeSSVisitor](#)

The Visitor that finds unhandled exceptions

Field Summary

private java.lang.String	errorMessage
-----------------------------	------------------------------

Fields inherited from class edu.colby.cs.JeSS.util.[JeSSVisitor](#)

Fields inherited from class org.eclipse.jdt.core.dom.ASTVisitor

Constructor Summary

[ExceptionFinder](#)([VisitorManager](#) vManager)

Method Summary

boolean	visit (org.eclipse.jdt.core.dom.CatchClause node) Check to see if the catch clause has a block that has no statements
---------	--

Methods inherited from class edu.colby.cs.JeSS.util.[JeSSVisitor](#)

[parseClassName](#), [parseStandardName](#), [reportProblem](#)

Methods inherited from class org.eclipse.jdt.core.dom.ASTVisitor

endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,
endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,
endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,
endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.visitors

Class InnerClassFinder

```
java.lang.Object
  extended by org.eclipse.jdt.core.dom.ASTVisitor
    extended by edu.colby.cs.JeSS.util.JeSSVisitor
      extended by edu.colby.cs.JeSS.visitors.InnerClassFinder
```

public class **InnerClassFinder**

extends [JeSSVisitor](#)

The visitor to find inner classes

Field Summary

private java.lang.String	errorMessage
-----------------------------	------------------------------

Fields inherited from class edu.colby.cs.JeSS.util.[JeSSVisitor](#)

Fields inherited from class org.eclipse.jdt.core.dom.ASTVisitor

Constructor Summary

[InnerClassFinder](#)([VisitorManager](#) vManager)

Method Summary

boolean	visit (org.eclipse.jdt.core.dom AnonymousClassDeclaration node) Report problem when a Declaration of this type is found
---------	--

Methods inherited from class edu.colby.cs.JeSS.util.[JeSSVisitor](#)

[parseClassName](#), [parseStandardName](#), [reportProblem](#)

Methods inherited from class org.eclipse.jdt.core.dom.ASTVisitor

endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,
endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,
endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,
endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV CLASS [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.actions

Class JeSSScanAction

java.lang.Object
 extended by edu.colby.cs.JeSS.actions.JeSSScanAction

All Implemented Interfaces:

org.eclipse.ui.IActionDelegate, org.eclipse.ui.IWorkbenchWindowActionDelegate

public class **JeSSScanAction**
 extends java.lang.Object
 implements org.eclipse.ui.IWorkbenchWindowActionDelegate

This class tracks the selection of the current user environment and sends the selected items along to the JeSSClearingHouse to determine their type and proper processing path for a Security Audit

See Also:

IWorkbenchWindowActionDelegate

Nested Class Summary

private class	JeSSScanAction.JeSSRunnable
------------------	---

Field Summary

private org.eclipse.ui.IWorkbenchWindow	window
private org.eclipse.ui.IWorkbenchPart	workbenchPart

Constructor Summary

[JeSSScanAction](#)()
 The constructor - does Nothing

Method Summary

void	dispose () Eclipse Generated Code - Does Nothing
private org.eclipse.jface.viewers.StructuredSelection	getStructuredSelection () Uses the current active workbench part to determine the

		selected objects.
<code>private org.eclipse.ui.IWorkbenchPart</code>		getWorkbenchPart()
	<code>void</code>	init (org.eclipse.ui.IWorkbenchWindow window) Caches window object and calls refreshActivePart() to s currently active workbench part
	<code>private void</code>	refreshActivePart () Stores the currently active workbench part in the local v workbenchPart
	<code>void</code>	run (org.eclipse.jface.action.IAction action) Sets up a runnable action so all resource changes are gr together when the scan is run and show the progress of the sca the status bar on the workbench window
	<code>private void</code>	runJeSS (org.eclipse.core.runtime.IProgressMonitor m The current selection is then determined and the objects comprise that selection are passed along to the JeSSClearingH one at a time to determine their type and proper processing p
	<code>void</code>	selectionChanged (org.eclipse.jface.action.IAction a org.eclipse.jface.viewers.ISelection selection) Eclipse Generated Code - Does Nothing
	<code>private void</code>	showMessage (java.lang.String message) A helper method to display a "JeSS Plug-in" titled mess dialog with the passed string as the message

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

window

`private org.eclipse.ui.IWorkbenchWindow window`

workbenchPart

`private org.eclipse.ui.IWorkbenchPart workbenchPart`

Constructor Detail

JeSSScanAction

`public JeSSScanAction()`

The constructor - does Nothing

Method Detail

run

```
public void run(org.eclipse.jface.action.IAction action)
```

Sets up a runnable action so all resource changes are grouped together when the scan is run and show the progress of the scan using the status bar on the workbench window

Specified by:

run in interface `org.eclipse.ui.IActionDelegate`

See Also:

`IActionDelegate.run(org.eclipse.jface.action.IAction)`

runJeSS

```
private void runJeSS(org.eclipse.core.runtime.IProgressMonitor monitor)
```

The current selection is then determined and the objects that comprise that selection are passed along to the JeSSClearingHouse one at a time to determine their type and proper processing path. Error messages are returned if there is an invalid selection

Parameters:

`monitor` - - the progress monitor to be used to display the progress of the scan

showMessage

```
private void showMessage(java.lang.String message)
```

A helper method to display a "JeSS Plug-in" titled message dialog with the passed string as the message

Parameters:

`message` - - the string to be displayed

getWorkbenchPart

```
private org.eclipse.ui.IWorkbenchPart getWorkbenchPart()
```

Returns:

`workbenchPart` - current active `IWorkbenchPart`

getStructuredSelection

```
private org.eclipse.jface.viewers.StructuredSelection getStructuredSelection()
```

Uses the current active workbench part to determine the current selected objects. If nothing is currently selected or a text editor is currently open (and text is selected) then null is returned.

Returns:

StructuredSelection - see [org.eclipse.jface.viewers.StructuredSelection](#)

init

```
public void init(org.eclipse.ui.IWorkbenchWindow window)
```

Caches window object and calls `refreshActivePart()` to store the currently active workbench part

Specified by:

`init` in interface `org.eclipse.ui.IWorkbenchWindowActionDelegate`

See Also:

`IWorkbenchWindowActionDelegate.init(org.eclipse.ui.IWorkbenchWindow)`

refreshActivePart

```
private void refreshActivePart()
```

Stores the currently active workbench part in the local variable `workbenchPart`

selectionChanged

```
public void selectionChanged(org.eclipse.jface.action.IAction action,  
                             org.eclipse.jface.viewers.ISelection selection)
```

Eclipse Generated Code - Does Nothing

Specified by:

`selectionChanged` in interface `org.eclipse.ui.IActionDelegate`

See Also:

`IActionDelegate.selectionChanged(org.eclipse.jface.action.IAction, org.eclipse.jface.viewers.ISelection)`

dispose

```
public void dispose()
```

Eclipse Generated Code - Does Nothing

Specified by:

`dispose` in interface `org.eclipse.ui.IWorkbenchWindowActionDelegate`

See Also:

`IWorkbenchWindowActionDelegate.dispose()`

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.actions

Class JeSSClearingHouse

java.lang.Object
 extended by edu.colby.cs.JeSS.actions.JeSSClearingHouse

public class **JeSSClearingHouse**
 extends java.lang.Object

This class acts as a clearing house to determine the type of the selected object and then pass it along the proper path to be processed. Each "process" method strips all previous JeSS markers off of the element and then creates a new instance of SecurityScanner to perform the scan

Author:

rspitler

Constructor Summary

[JeSSClearingHouse\(\)](#)

Method Summary

private static int	processCompilationUnit (org.eclipse.jdt.core.ICompilationUnit unit, org.eclipse.jface.preference.IPreferenceStore store) Process a CompilationUnit to be scanned
private static int	processJavaProject (org.eclipse.jdt.core.IJavaProject project, org.eclipse.jface.preference.IPreferenceStore store) Process a javaProject to be scanned
static int	processObject (java.lang.Object obj, org.eclipse.jface.preference.IPreferenceStore store) The initial screening that determines the type of the selected object
private static int	processPackageFragment (org.eclipse.jdt.core.IPackageFragment fragment, org.eclipse.jface.preference.IPreferenceStore store) Process a PackageFragment to be scanned
private static int	processPackageFragmentRoot (org.eclipse.jdt.core.IPackageFragmentRoot root, org.eclipse.jface.preference.IPreferenceStore store) Process a package fragment root

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

JeSSClearingHouse

```
public JeSSClearingHouse()
```

Method Detail

processObject

```
public static int processObject(java.lang.Object obj,  
                               org.eclipse.jface.preference.IPreferenceStore store)
```

The initial screening that determines the type of the selected object

Parameters:

obj -- the selected object to be processed
store -- the preferences that determine the extent of the scan

Returns:

warningsCount - the number of security bugs found

processCompilationUnit

```
private static int processCompilationUnit(org.eclipse.jdt.core.ICompilationUnit unit,  
                                         org.eclipse.jface.preference.IPreferenceStore store)
```

Process a CompilationUnit to be scanned

Parameters:

unit -- the CompilationUnit to be scanned
store -- the preferences for the scan

Returns:

warningsCount - the number of bugs found

processPackageFragment

```
private static int processPackageFragment(org.eclipse.jdt.core.IPackageFragment fragment,  
                                         org.eclipse.jface.preference.IPreferenceStore store)
```

Process a PackageFragment to be scanned

Parameters:

fragment -- the package fragment to be scanned
store -- the preferences for the scan

Returns:

warningsCount - the number of bugs found

processPackageFragmentRoot

```
private static int processPackageFragmentRoot(org.eclipse.jdt.core.IPackageFragmentRoot root,  
                                             org.eclipse.jface.preference.IPreferenceStore store)
```

Process a package fragment root

Parameters:

root - - the package root to be processed

store - - the preferences for the scan

Returns:

warningscount - the number of bugs found

processJavaProject

```
private static int processJavaProject(org.eclipse.jdt.core.IJavaProject project,  
                                     org.eclipse.jface.preference.IPreferenceStore store)
```

Process a javaProject to be scanned

Parameters:

project - - the project to be scanned

store - - the preferences for the scan

Returns:

warningsCount - the number of bugs found

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.scanner

Class VisitorManager

java.lang.Object
 extended by `edu.colby.cs.JeSS.scanner.VisitorManager`

public class **VisitorManager**

extends java.lang.Object

This class is the root of the security analysis. It accepts the IPreference store in its initialization to determine the types of scans being performed. The scan() method then accepts the root of an AST and the corresponding source file for error reporting. This is the only class in the JeSS plugin that deals with markers. All markers are created and deleted from methods in this class. Errors are reported from the visitors through the reportProblem() method of JeSSVisitor, which in turn calls the reportProblem() method of this class.

Author:

rspitler

Field Summary

static java.lang.String	MARKER_ID
private int	problems
private org.eclipse.core.resources.IFile	source
private JeSSVisitorCollection	visitors

Constructor Summary

[VisitorManager](#)(org.eclipse.jface.preference.IPreferenceStore store)

Initialize the class using the IPreferenceStore to determine what scanners are used during the security analysis

Method Summary

static void	deleteMarker (org.eclipse.core.resources.IMarker marker) Delete the passed marker.
static boolean	deleteSecurityMarkers (org.eclipse.core.resources.IResource source) This method deletes all of the security markers in the passed resource
org.eclipse.core.resources.IFile	getResource ()

void	reportProblem (java.lang.String errorMessage, Location loc, boolean isError, java.lang.String errorType) Take the passed values and create a Marker on the resource stored by the VisitorManager.
int	scan (org.eclipse.jdt.core.dom.CompilationUnit result, org.eclipse.core.resources.IFile file) Accept an AST root node in the form org.eclipse.jdt.core.dom.CompilationUnit and scan the AST for security bugs.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

problems

```
private int problems
```

MARKER_ID

```
public static java.lang.String MARKER_ID
```

visitors

```
private JeSSVisitorCollection visitors
```

source

```
private org.eclipse.core.resources.IFile source
```

Constructor Detail

VisitorManager

```
public VisitorManager(org.eclipse.jface.preference.IPreferenceStore store)
```

Initialize the class using the IPreferenceStore to determine what scanners are used during the security analysis

Parameters:

store - - the plugin preferences

Method Detail

scan

```
public int scan(org.eclipse.jdt.core.dom.CompilationUnit result,  
                org.eclipse.core.resources.IFile file)
```

Accept an AST root node in the form org.eclipse.jdt.core.dom.CompilationUnit and scan the AST for security bugs.

Parameters:

`result` -- the AST to be scanned
`file` -- the local resource for the AST

Returns:

`warningsCount` - the number of warnings generated

getResource

```
public org.eclipse.core.resources.IFile getResource()
```

Returns:

`source` - the underlying file of the current AST

reportProblem

```
public void reportProblem(java.lang.String errorMessage,  
                           Location loc,  
                           boolean isError,  
                           java.lang.String errorType)
```

Take the passed values and create a Marker on the resource stored by the VisitorManager. Use MarkerUtilities to ensure that the marker appears in the source.

Parameters:

`errorMessage` -- the message related to the security bug
`loc` -- the Location object storing the placement of the bug
`isError` -- a boolean to determine if the bug is an error or warning
`errorType` -- TODO for future implementation of error resolution

deleteMarker

```
public static void deleteMarker(org.eclipse.core.resources.IMarker marker)
```

Delete the passed marker. This method is included here as it is the only way a marker is deleted in JeSS. If future implementations require additional handling when markers are removed then this is where the changes would be made

Parameters:

`marker` -- the marker to be removed

deleteSecurityMarkers

```
public static boolean deleteSecurityMarkers(org.eclipse.core.resources.IResource source)
```

This method deletes all of the security markers in the passed resource

Parameters:

source - - tje source in which to remove the markers

Returns:

boolean - result depending upon success

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.util

Class VectorVisitorCollection

java.lang.Object

extended by `edu.colby.cs.JeSS.util.VectorVisitorCollection`

All Implemented Interfaces:

[JeSSVisitorCollection](#)

public class **VectorVisitorCollection**

extends java.lang.Object

implements [JeSSVisitorCollection](#)

The current implementation of JeSSVisitorCollection. This is done using a Vector as a delegate.

Field Summary

private java.util.Vector	collection
-----------------------------	----------------------------

Constructor Summary

[VectorVisitorCollection](#)()

Method Summary

void	add (JeSSVisitor visitor) Add another visitor to the Collection
void	append (JeSSVisitorCollection otherJeSSVC) Add all visitors from another Collection of this type
JeSSVisitor	elementAt (int i) Retrieve an element at specified location
java.util.Collection	getCollection () A helper method for append.
int	size ()

Methods inherited from class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Field Detail

collection

```
private java.util.Vector collection
```

Constructor Detail

VectorVisitorCollection

```
public VectorVisitorCollection()
```

Method Detail

elementAt

```
public JeSSVisitor elementAt(int i)
```

Description copied from interface: [JeSSVisitorCollection](#)

Retrieve an element at specified location

Specified by:

[elementAt](#) in interface [JeSSVisitorCollection](#)

Parameters:

i -- the location of the desired element

Returns:

JeSSVisitor - the JeSSVisitor at i

add

```
public void add(JeSSVisitor visitor)
```

Description copied from interface: [JeSSVisitorCollection](#)

Add another visitor to the Collection

Specified by:

[add](#) in interface [JeSSVisitorCollection](#)

Parameters:

visitor -- the visitor to be added

append

```
public void append(JeSSVisitorCollection otherJeSSVC)
```

Description copied from interface: [JeSSVisitorCollection](#)

Add all visitors from another Collection of this type

Specified by:

[append](#) in interface [JeSSVisitorCollection](#)

Parameters:

`otherJeSSVC` - - the collection to be added

getCollection

```
public java.util.Collection getCollection()
```

Description copied from interface: [JeSSVisitorCollection](#)

A helper method for `append`. This allows any implementations of this interface to use any type of delegate so long as it is a collection

Specified by:

[getCollection](#) in interface [JeSSVisitorCollection](#)

Returns:

Collection - the underlying delegate collection of an implementation of `JeSSVisitorCollection`

size

```
public int size()
```

Specified by:

[size](#) in interface [JeSSVisitorCollection](#)

Returns:

the number of elements in the collection

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.actions

Class ShowPreferenceAction

java.lang.Object

extended by `edu.colby.cs.JeSS.actions.ShowPreferenceAction`

All Implemented Interfaces:

org.eclipse.ui.IActionDelegate, org.eclipse.ui.IWorkbenchWindowActionDelegate

public class **ShowPreferenceAction**

extends java.lang.Object

implements org.eclipse.ui.IWorkbenchWindowActionDelegate

Field Summary

private org.eclipse.ui.IWorkbenchWindow	window
--	------------------------

Constructor Summary

[ShowPreferenceAction](#)()

The constructor.

Method Summary

void	dispose () Eclipse Generated Code - Does Nothing
private org.eclipse.swt.widgets.Shell	getShell ()
void	init (org.eclipse.ui.IWorkbenchWindow window) Store the Workbench window for later use
void	run (org.eclipse.jface.action.IAction action) Create a new Preference Manager showing the JeSSPreferencePage
void	selectionChanged (org.eclipse.jface.action.IAction action, org.eclipse.jface.viewers.ISelection selection) Eclipse Generated Code - Does Nothing

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

window

```
private org.eclipse.ui.IWorkbenchWindow window
```

Constructor Detail

ShowPreferenceAction

```
public ShowPreferenceAction()
```

The constructor.

Method Detail

run

```
public void run(org.eclipse.jface.action.IAction action)
```

Create a new Preference Manager showing the JeSSPreferencePage

Specified by:

run in interface org.eclipse.ui.IActionDelegate

See Also:

IActionDelegate.run(org.eclipse.jface.action.IAction)

getShell

```
private org.eclipse.swt.widgets.Shell getShell()
```

Returns:

Shell

init

```
public void init(org.eclipse.ui.IWorkbenchWindow window)
```

Store the Workbench window for later use

Specified by:

init in interface org.eclipse.ui.IWorkbenchWindowActionDelegate

Parameters:

window - - the current workbench window

See Also:

```
IWorkbenchWindowActionDelegate.init(org.eclipse.ui.IWorkbenchWindow)
```

selectionChanged

```
public void selectionChanged(org.eclipse.jface.action.IAction action,  
                             org.eclipse.jface.viewers.ISelection selection)
```

Eclipse Generated Code - Does Nothing

Specified by:

selectionChanged in interface org.eclipse.ui.IActionDelegate

See Also:

IActionDelegate.selectionChanged(org.eclipse.jface.action.IAction,
org.eclipse.jface.viewers.ISelection)

dispose

```
public void dispose()
```

Eclipse Generated Code - Does Nothing

Specified by:

dispose in interface org.eclipse.ui.IWorkbenchWindowActionDelegate

See Also:

IWorkbenchWindowActionDelegate.dispose()

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

Overview [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)PREV CLASS [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.scanner

Class SecurityScannerjava.lang.Object
extended by `edu.colby.cs.JeSS.scanner.SecurityScanner`public class **SecurityScanner**
extends java.lang.Object

This class receives a java Project, Package or Compilation Unit and generates an AST. This AST is then relayed to the VisitorManager class where it is evaluated for security bugs. This process passes the integer warningsCount to track the number of bugs found.

Author:

rspitler

Field Summary

private VisitorManager	vManager
---	--------------------------

Constructor Summary[SecurityScanner](#)(org.eclipse.jface.preference.IPreferenceStore store)**Method Summary**

private int	scanCompilationUnit (org.eclipse.jdt.core.dom.CompilationUnit result, org.eclipse.core.resources.IFile file) Processes a org.eclipse.jdt.core.dom.CompilationUnit which is the root of an AST and passes it to the VisitorManager for security analysis
int	scanCompilationUnit (org.eclipse.jdt.core.ICompilationUnit compUnit) WARNING: do not confuse with the private method of the same name! This method accepts type org.eclipse.jdt.core.ICompilationUnit and then extracts the local resource and generates an AST before it is passed along for security analysis
int	scanPackage (org.eclipse.jdt.core.IPackageFragment fragment) Receive a package fragment and extract the compilation units to be processed individually.
int	scanProject (org.eclipse.jdt.core.IJavaProject project) This method receives a Java project and the projects immediate Resource.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail**vManager**private [VisitorManager](#) **vManager****Constructor Detail****SecurityScanner**

```
public SecurityScanner(org.eclipse.jface.preference.IPreferenceStore store)
```

Method Detail

scanProject

```
public int scanProject(org.eclipse.jdt.core.IJavaProject project)
    throws org.eclipse.jdt.core.JavaModelException
```

This method receives a Java project and the projects immediate Resource. It then extracts the package fragments from the project and calls a helper method to extract the compilation Units from the packages.

Parameters:

project -- the project to be scanned

Returns:

warningsCount - number of bugs found in the project

Throws:

org.eclipse.jdt.core.JavaModelException

scanPackage

```
public int scanPackage(org.eclipse.jdt.core.IPackageFragment fragment)
    throws org.eclipse.jdt.core.JavaModelException
```

Receive a package fragment and extract the compilation units to be processed individually.

Parameters:

fragment -- the fragment to be processed

Returns:

warningsCount - the number of bugs found

Throws:

org.eclipse.jdt.core.JavaModelException

scanCompilationUnit

```
public int scanCompilationUnit(org.eclipse.jdt.core.ICompilationUnit compUnit)
```

WARNING: do not confuse with the private method of the same name! This method accepts type org.eclipse.jdt.core.ICompilationUnit and then extracts the local resource and generates an AST before it is passed along for security analysis

Parameters:

compUnit -- the ICompilationUnit to be analyzed

Returns:

warningsCount - the number of bugs found in this source

scanCompilationUnit

```
private int scanCompilationUnit(org.eclipse.jdt.core.dom.CompilationUnit result,
    org.eclipse.core.resources.IFile file)
```

Processes a org.eclipse.jdt.core.dom.CompilationUnit which is the root of an AST and passes it to the VisitorManager for security analysis

Parameters:

result - the root of the AST to be analyzed

file - the local resource of the CompilationUnit

Returns:

The number of bugs found

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV CLASS [NEXT CLASS](#)

SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.visitors

Class PublicModifierFinder

```

java.lang.Object
  extended by org.eclipse.jdt.core.dom.ASTVisitor
    extended by edu.colby.cs.JeSS.util.JeSSVisitor
      extended by edu.colby.cs.JeSS.visitors.PublicModifierFinder

```

public class **PublicModifierFinder**extends [JeSSVisitor](#)

This class finds all publicly declared Fields and Methods

Field SummaryFields inherited from class edu.colby.cs.JeSS.util.[JeSSVisitor](#)

Fields inherited from class org.eclipse.jdt.core.dom.ASTVisitor

Constructor Summary[PublicModifierFinder](#)([VisitorManager](#) vManager)**Method Summary**

boolean	visit (org.eclipse.jdt.core.dom.FieldDeclaration node) Check the Field declaration to see if it is public
---------	--

boolean	visit (org.eclipse.jdt.core.dom.MethodDeclaration node) Check the Method declaration to see if it is public
---------	--

Methods inherited from class edu.colby.cs.JeSS.util.[JeSSVisitor](#)[parseClassName](#), [parseStandardName](#), [reportProblem](#)

Methods inherited from class org.eclipse.jdt.core.dom.ASTVisitor

```

endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,
endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,
endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,
endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,

```



```
endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit, endVisit,
endVisit, endVisit, postVisit, preVisit, visit, visit, visit, visit, visit, visit, visit,
visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit,
visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit,
visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit,
visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit,
visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit,
visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit,
visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit, visit,
visit, visit, visit
```

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait,
wait
```

Constructor Detail

PublicFinalClassFinder

```
public PublicFinalClassFinder(VisitorManager vManager)
```

Method Detail

visit

```
public boolean visit(org.eclipse.jdt.core.dom.TypeDeclaration node)
```

Check the Type declaration to determine if the class is public or not Final

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.visitors

Class ProtectedModifierFinder

java.lang.Object

extended by org.eclipse.jdt.core.dom.ASTVisitor

extended by [edu.colby.cs.JeSS.util.JeSSVisitor](#)

extended by [edu.colby.cs.JeSS.visitors.ProtectedModifierFinder](#)

public class **ProtectedModifierFinder**

extends [JeSSVisitor](#)

This class finds Fields and Mehtods with the protected modifier

Field Summary

Fields inherited from class edu.colby.cs.JeSS.util.[JeSSVisitor](#)

Fields inherited from class org.eclipse.jdt.core.dom.ASTVisitor

Constructor Summary

[ProtectedModifierFinder](#)([VisitorManager](#) vManager)

Method Summary

boolean	visit (org.eclipse.jdt.core.dom.FieldDeclaration node) Check the Field declaration for a protected modifier
boolean	visit (org.eclipse.jdt.core.dom.MethodDeclaration node) Check the method declaration for a protected modifier
boolean	visit (org.eclipse.jdt.core.dom.TypeDeclaration node) Check the Type declaration for a protected modifier

Methods inherited from class edu.colby.cs.JeSS.util.[JeSSVisitor](#)

[parseClassName](#), [parseStandardName](#), [reportProblem](#)

Methods inherited from class org.eclipse.jdt.core.dom.ASTVisitor

SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.util

Class PluginLoader.JeSSPluginLoader

java.lang.Object

extended by java.lang.ClassLoader

extended by edu.colby.cs.JeSS.util.PluginLoader.JeSSPluginLoader

Enclosing class:

[PluginLoader](#)

private class **PluginLoader.JeSSPluginLoader**

extends java.lang.ClassLoader

This is a private ClassLoader that loads a class from file on disk. This class uses a delegate to perform the loading operations.

Nested Class Summary

Nested classes inherited from class java.lang.ClassLoader

Field Summary

Fields inherited from class java.lang.ClassLoader

Constructor Summary

[PluginLoader.JeSSPluginLoader](#)(java.lang.ClassLoader delegate)

Create the class loader using the passed delegate

Method Summary

java.lang.Class

[defineClass](#)(java.io.File file)

Create a class object from the passed class file

Methods inherited from class java.lang.ClassLoader

clearAssertionStatus, defineClass, defineClass, defineClass, definePackage, findClass, findLibrary, findLoadedClass, findResource, findResources, findSystemClass, getPackage, getPackages, getParent, getResource, getResourceAsStream, getResources, getSystemClassLoader, getSystemResource, getSystemResourceAsStream, getSystemResources, loadClass, loadClass, resolveClass, setClassAssertionStatus, setDefaultAssertionStatus, setPackageAssertionStatus, setSigners

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PluginLoader.JeSSPluginLoader

```
public PluginLoader.JeSSPluginLoader(java.lang.ClassLoader delegate)
```

Create the class loader using the passed delegate

Parameters:

delegate -

Method Detail

defineClass

```
public java.lang.Class defineClass(java.io.File file)
```

Create a class object from the passed class file

Parameters:

file - - the class file to be turned into a Class object

Returns:

Class - the Class object for the file

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.util

Class PluginLoader

java.lang.Object
 extended by `edu.colby.cs.JeSS.util.PluginLoader`

public class **PluginLoader**
 extends java.lang.Object

This class is responsible for dynamically loading the external visitors used in a JeSS scan. The class uses the path as set in the JeSS preferences to search for the external visitors. The type is checked and the existence of a proper constructor is confirmed. Then the external visitors are all added to a JeSSVisitorCollection and returned.

Nested Class Summary

private class	PluginLoader.classFileFilter The FileFilter that is used to screen out non-class files
private class	PluginLoader.JeSSPluginLoader This is a private ClassLoader that loads a class from file on disk.

Field Summary

private java.lang.Class[]	args
---------------------------	----------------------

Constructor Summary

[PluginLoader](#)()

Method Summary

JeSSVisitorCollection	getPluginVisitors (org.eclipse.jface.preference.IPreferenceStore store, VisitorManager vManager) This method searches the JeSS plugin folder for class files it then checks to make sure the class files are valid JeSSVisitors and then instantiates them and passes them back in a JeSSVisitorCollection
private boolean	isValidJeSSVisitor (java.lang.Class userClass) Check to see if the loaded object is in fact a valid JeSSVisitor.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

args

```
private java.lang.Class[] args
```

Constructor Detail

PluginLoader

```
public PluginLoader()
```

Method Detail

getPluginVisitors

```
public JeSSVisitorCollection getPluginVisitors(org.eclipse.jface.preference.IPreferenceStore store,
VisitorManager vManager)
```

This method searches the JeSS plugin folder for class files it then checks to make sure the class files are valid JeSSVisitors and then instantiates them and passes them back in a JeSSVisitorCollection

Returns:

JeSSVisitorCollection - the collection of discovered visitors

isValidJeSSVisitor

```
private boolean isValidJeSSVisitor(java.lang.Class userClass)
```

Check to see if the loaded object is in fact a valid JeSSVisitor. As in, it is a subclass of JeSSVisitor and it has a constructor that takes type VisitorManager as a paramter

Parameters:

userClass - - the object to be tested

Returns:

boolean - true if a valid visitor

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#)

DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.util

Class PluginLoader.classFileFilter

java.lang.Object

extended by `edu.colby.cs.JeSS.util.PluginLoader.classFileFilter`

All Implemented Interfaces:

java.io.FileFilter

Enclosing class:

[PluginLoader](#)

private class **PluginLoader.classFileFilter**

extends java.lang.Object

implements java.io.FileFilter

The FileFilter that is used to screen out non-class files

Constructor Summary

private	PluginLoader.classFileFilter ()
---------	---

Method Summary

boolean	accept (java.io.File file) Retrieve the file name and check to see if it is a class file
---------	---

Methods inherited from class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor Detail

PluginLoader.classFileFilter

```
private PluginLoader.classFileFilter()
```

Method Detail

accept

```
public boolean accept(java.io.File file)
```

Retrieve the file name and check to see if it is a class file

Specified by:

accept in interface `java.io.FileFilter`

Returns:

boolean - true if it is a class file

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#)DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.visitors

Class MissingModifierFinder

```

java.lang.Object
  extended by org.eclipse.jdt.core.dom.ASTVisitor
    extended by edu.colby.cs.JeSS.util.JeSSVisitor
      extended by edu.colby.cs.JeSS.visitors.MissingModifierFinder

```

public class **MissingModifierFinder**extends [JeSSVisitor](#)

The visitor that finds Types, Fields, and Methods that are missing modifiers.

Field SummaryFields inherited from class edu.colby.cs.JeSS.util.[JeSSVisitor](#)

Fields inherited from class org.eclipse.jdt.core.dom.ASTVisitor

Constructor Summary[MissingModifierFinder](#)([VisitorManager](#) vManager)**Method Summary**

boolean	visit (org.eclipse.jdt.core.dom.FieldDeclaration node) Check the Field declaration for a missing modifier
boolean	visit (org.eclipse.jdt.core.dom.MethodDeclaration node) Check the Method declaration for a missing modifier
boolean	visit (org.eclipse.jdt.core.dom.TypeDeclaration node) Check the Type declaration for a missing modifier

Methods inherited from class edu.colby.cs.JeSS.util.[JeSSVisitor](#)[parseClassName](#), [parseStandardName](#), [reportProblem](#)

Methods inherited from class org.eclipse.jdt.core.dom.ASTVisitor

SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.util

Class MarkerResolutionGenerator

java.lang.Object

extended by edu.colby.cs.JeSS.util.MarkerResolutionGenerator

All Implemented Interfaces:

org.eclipse.ui.IMarkerResolutionGenerator

public class **MarkerResolutionGenerator**

extends java.lang.Object

implements org.eclipse.ui.IMarkerResolutionGenerator

This class provides the implementation for simple resolutions of JeSS security bugs. It is possible to ignore the individual marker, ignore all markers on a particular file, and to ignore all markers on a project. Further implementation of Error resolutions should be done in this class.

Author:

rspitler

Nested Class Summary

private class	MarkerResolutionGenerator.JeSSMarkerResolutionIgnore The resolution that allows the deletion of a single JeSS marker
private class	MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllFile The resolution that allows the deletion of all markers in the containing file
private class	MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllProject The resolution that allows the deletion of all markers in the containing project

Constructor Summary

[MarkerResolutionGenerator](#)()

Method Summary

org.eclipse.ui.IMarkerResolution[]	getResolutions (org.eclipse.core.resources.IMarker marker) This method returns the array of possible markers resolutions for the given marker.
boolean	hasResolutions (org.eclipse.core.resources.IMarker marker) A boolean check to see if resolutions exist for the marker

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MarkerResolutionGenerator

```
public MarkerResolutionGenerator()
```

Method Detail

getResolutions

```
public org.eclipse.ui.IMarkerResolution[] getResolutions(org.eclipse.core.resources.IMarker marker)
```

This method returns the array of possible markers resolutions for the given marker.

Specified by:

getResolutions in interface org.eclipse.ui.IMarkerResolutionGenerator

Parameters:

marker -- the marker to find resolutions for

Returns:

an array containing resolutions for the marker

See Also:

IMarkerResolutionGenerator.getResolutions(org.eclipse.core.resources.IMarker)

hasResolutions

```
public boolean hasResolutions(org.eclipse.core.resources.IMarker marker)
```

A boolean check to see if resolutions exist for the marker

Parameters:

marker -- the marker to check for resolutions

Returns:

boolean - depending upon existence of resolutions

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.util

Class Location

java.lang.Object
 extended by `edu.colby.cs.JeSS.util.Location`

public class **Location**
 extends java.lang.Object

This class stores the information about a problem to be used in marker creation

Field Summary

private int	charEnd
private int	charStart
private org.eclipse.core.resources.IFile	file
private java.lang.String	key
private int	lineNumber

Constructor Summary

[Location](#)()

Method Summary

int	getCharEnd ()
int	getCharStart ()
org.eclipse.core.resources.IFile	getFile ()
java.lang.String	getKey ()

int	getLineNumber()
void	setCharEnd(int charEnd)
void	setCharStart(int charStart)
void	setFile(org.eclipse.core.resources.IFile file)
void	setKey(java.lang.String key)
void	setLineNumber(int lineNumber)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

file

```
private org.eclipse.core.resources.IFile file
```

key

```
private java.lang.String key
```

charStart

```
private int charStart
```

charEnd

```
private int charEnd
```

lineNumber

```
private int lineNumber
```

Constructor Detail

Location

```
public Location()
```

Method Detail

getLineNumber

```
public int getLineNumber()
```

Returns:

lineNumber - the line number of the problem.

setLineNumber

```
public void setLineNumber(int lineNumber)
```

Parameters:

lineNumber - The lineNumber to set.

getCharEnd

```
public int getCharEnd()
```

Returns:

charEnd - the last character of the problem in the file

setCharEnd

```
public void setCharEnd(int charEnd)
```

Parameters:

charEnd - The charEnd to set.

getCharStart

```
public int getCharStart()
```

Returns:

Returns the charStart.

setCharStart

```
public void setCharStart(int charStart)
```

Parameters:

charStart - The charStart to set.

getFile

```
public org.eclipse.core.resources.IFile getFile()
```

Returns:

Returns the file.

setFile

```
public void setFile(org.eclipse.core.resources.IFile file)
```

Parameters:

file - The file to set.

getKey

```
public java.lang.String getKey()
```

Returns:

Returns the key.

setKey

```
public void setKey(java.lang.String key)
```

Parameters:

key - The key to set.

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.util

Class

MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllProject

java.lang.Object

extended by `edu.colby.cs.JeSS.util.MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllProject`

All Implemented Interfaces:

org.eclipse.ui.IMarkerResolution

Enclosing class:

[MarkerResolutionGenerator](#)

private class **MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllProject**

extends java.lang.Object

implements org.eclipse.ui.IMarkerResolution

The resolution that allows the deletion of all markers in the containing the project

Constructor Summary

private	MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllProject()
---------	--

Method Summary

java.lang.String	getLabel()
void	run(org.eclipse.core.resources.IMarker marker)

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllProject

```
private MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllProject()
```

Method Detail

getLabel

```
public java.lang.String getLabel()
```

Specified by:

getLabel in interface org.eclipse.ui.IMarkerResolution

run

```
public void run(org.eclipse.core.resources.IMarker marker)
```

Specified by:

run in interface org.eclipse.ui.IMarkerResolution

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#) [All Classes](#)SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.preferences

Class JeSSPreferencePage

java.lang.Object

extended by org.eclipse.jface.dialogs.DialogPage

extended by org.eclipse.jface.preference.PreferencePage

extended by org.eclipse.jface.preference.FieldEditorPreferencePage

extended by **edu.colby.cs.JeSS.preferences.JeSSPreferencePage**

All Implemented Interfaces:

java.util.EventListener, org.eclipse.jface.dialogs.IDialogPage, org.eclipse.jface.dialogs.IMessageProvider,

org.eclipse.jface.preference.IPreferencePage, org.eclipse.jface.util.IPropertyChangeListener,

org.eclipse.ui.IWorkbenchPreferencePage

public class **JeSSPreferencePage**

extends org.eclipse.jface.preference.FieldEditorPreferencePage

implements org.eclipse.ui.IWorkbenchPreferencePage

This class represents a preference page that is contributed to the Preferences dialog. The built in visitors are referenced in this preference page. They are represented as a boolean checkbox. There is also the option for selecting the directory that JeSS searches in to discover user defined visitors. The default for this directory is the "plugins" folder in the edu.colby.cs.JeSS plugin

Field Summary

static java.lang.String	EXCEPTIONS
static java.lang.String	EXTERNAL_VISITORS
static java.lang.String	FINAL
static java.lang.String	INNER_CLASS
static java.lang.String	MISS_MODIFIER
static java.lang.String	PLUGIN_DIR
static java.lang.String	PROTECT_MODIFIER
static java.lang.String	PUBLIC_FINAL_CLASS

static java.lang.String	PUBLIC_MODIFIER
-------------------------	---------------------------------

Fields inherited from class org.eclipse.jface.preference.FieldEditorPreferencePage

FLAT, GRID, MARGIN_HEIGHT, MARGIN_WIDTH, VERTICAL_SPACING

Fields inherited from class org.eclipse.jface.preference.PreferencePage

Fields inherited from class org.eclipse.jface.dialogs.DialogPage

Fields inherited from interface org.eclipse.jface.dialogs.IMessageProvider

ERROR, INFORMATION, NONE, WARNING

Constructor Summary

[JeSSPreferencePage](#) ()

Method Summary

void	createFieldEditors () Creates the field editors.
void	init (org.eclipse.ui.IWorkbench workbench) Eclipse Generated - does nothing
private void	initializeDefaults () Set the default values of the preferences for the default scan.

Methods inherited from class org.eclipse.jface.preference.FieldEditorPreferencePage

addField, adjustGridLayout, applyFont, checkState, createContents, dispose, getFieldEditorParent, initialize, performDefaults, performOk, propertyChange, setVisible

Methods inherited from class org.eclipse.jface.preference.PreferencePage

applyDialogFont, computeSize, contributeButtons, createControl, createDescriptionLabel, createNoteComposite, doComputeSize, doGetPreferenceStore, getApplyButton, getContainer, getDefaultsButton, getPreferenceStore, isValid, noDefaultAndApplyButton, okToLeave, performApply, performCancel, performHelp, setContainer, setErrorMessage, setMessage, setPreferenceStore, setSize, setTitle, setValid, toString, updateApplyButton

Methods inherited from class org.eclipse.jface.dialogs.DialogPage

convertHeightInCharsToPixels, convertHorizontalDLUsToPixels, convertVerticalDLUsToPixels, convertWidthInCharsToPixels, getControl, getDescription, getDialogFontName, getErrorMessage, getFont, getImage, getMessage, getMessageType, getShell, getTitle, getToolTipText, initializeDialogUnits, isControlCreated, setButtonLayoutData, setControl, setDescription, setImageDescriptor, setMessage

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface org.eclipse.jface.preference.IPreferencePage

computeSize, isValid, okToLeave, performCancel, performOk, setContainer, setSize

Methods inherited from interface org.eclipse.jface.dialogs.IDialogPage

createControl, dispose, getControl, getDescription, getErrorMessage, getImage, getMessage, getTitle, performHelp, setDescription, setImageDescriptor, setTitle, setVisible

Field Detail

INNER_CLASS

```
public static final java.lang.String INNER_CLASS
```

See Also:

[Constant Field Values](#)

FINAL

```
public static final java.lang.String FINAL
```

See Also:

[Constant Field Values](#)

EXCEPTIONS

```
public static final java.lang.String EXCEPTIONS
```

See Also:

[Constant Field Values](#)

MISS_MODIFIER

```
public static final java.lang.String MISS_MODIFIER
```

See Also:

[Constant Field Values](#)

PROTECT_MODIFIER

```
public static final java.lang.String PROTECT_MODIFIER
```

See Also:

[Constant Field Values](#)

PUBLIC_MODIFIER

```
public static final java.lang.String PUBLIC_MODIFIER
```

See Also:

[Constant Field Values](#)

EXTERNAL_VISITORS

```
public static final java.lang.String EXTERNAL_VISITORS
```

See Also:

[Constant Field Values](#)

PUBLIC_FINAL_CLASS

```
public static final java.lang.String PUBLIC_FINAL_CLASS
```

See Also:

[Constant Field Values](#)

PLUGIN_DIR

```
public static final java.lang.String PLUGIN_DIR
```

See Also:

[Constant Field Values](#)

Constructor Detail

JeSSPreferencePage

```
public JeSSPreferencePage()
```

Method Detail

initializeDefaults

```
private void initializeDefaults()
```

Set the default values of the preferences for the default scan.

createFieldEditors

```
public void createFieldEditors()
```

Creates the field editors. Set up the boolean editors for the built in visitors and a directory selector to specify the source of the external editors

init

```
public void init(org.eclipse.ui.IWorkbench workbench)
```

Eclipse Generated - does nothing

Specified by:

`init` in interface `org.eclipse.ui.IWorkbenchPreferencePage`

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.util

Class JeSSVisitor

java.lang.Object
 extended by org.eclipse.jdt.core.dom.ASTVisitor
 extended by edu.colby.cs.JeSS.util.JeSSVisitor

Direct Known Subclasses:

[ExceptionFinder](#), [FinalFinder](#), [InnerClassFinder](#), [MissingModifierFinder](#), [ProtectedModifierFinder](#),
[PublicFinalClassFinder](#), [PublicModifierFinder](#)

public class **JeSSVisitor**
 extends org.eclipse.jdt.core.dom.ASTVisitor

This is the super class for all Visitors in the JeSS scanner. This is created to allow easy extensibility to the JeSS plugin. Simply create a sub-type of JeSSVisitor to find patterns in an AST and then use the reportProblem() method of JeSSVisitor to create a security marker.

Author:

Russell Spitler Mar 24, 2005

Field Summary

private VisitorManager	vManager
---	--------------------------

Fields inherited from class org.eclipse.jdt.core.dom.ASTVisitor

Constructor Summary

[JeSSVisitor](#)([VisitorManager](#) vManager)

The constructor stores a reference to the Visitor manager for error reporting

Method Summary

protected java.lang.String	parseClassName (java.lang.String string) This helper method parses a class name from the output of the standard toString() method in the TypeDeclaration AST node.
protected java.lang.String	parseStandardName (java.lang.String string) This method parses a user readable name from the toString() output of FieldDeclaration and MethodDeclaration.
void	reportProblem (org.eclipse.jdt.core.dom.ASTNode node, java.lang.String errorMessage) Used to report a problem and create a security marker for the security bug.

AST node. This method relies upon the standard format of TypeDeclaration[class CLASSNAME DECLARATIONS]. The name is converted to user readable form "class CLASSNAME"

Parameters:

`string` - - toString() from a TypeDeclaration AST node

Returns:

the name in user readable form

parseStandardName

```
protected java.lang.String parseStandardName(java.lang.String string)
```

This method parses a user readable name from the toString() output of FieldDeclaration and MethodDeclaration. This method relies on the standard format of `***Declaration[*** NAME]`

Parameters:

`string` - - toString() from a MethodDeclaration or FieldDeclaration node

Returns:

the name in user readable form

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV CLASS [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#) [All Classes](#)SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.util

Class**MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllFile**

java.lang.Object

extended by edu.colby.cs.JeSS.util.MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllFile

All Implemented Interfaces:

org.eclipse.ui.IMarkerResolution

Enclosing class:[MarkerResolutionGenerator](#)private class **MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllFile**

extends java.lang.Object

implements org.eclipse.ui.IMarkerResolution

The resolution that allows the deletion of all markers in the containing file

Constructor Summaryprivate [MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllFile](#) ()**Method Summary**java.lang.String [getLabel](#) ()void [run](#) (org.eclipse.core.resources.IMarker marker)**Methods inherited from class java.lang.Object**

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail**MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllFile**private **MarkerResolutionGenerator.JeSSMarkerResolutionIgnoreAllFile** ()**Method Detail**

getLabel

```
public java.lang.String getLabel()
```

Specified by:

`getLabel` in interface `org.eclipse.ui.IMarkerResolution`

run

```
public void run(org.eclipse.core.resources.IMarker marker)
```

Specified by:

`run` in interface `org.eclipse.ui.IMarkerResolution`

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#)

DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

edu.colby.cs.JeSS.util

Class MarkerResolutionGenerator.JeSSMarkerResolutionIgnore

java.lang.Object

extended by edu.colby.cs.JeSS.util.MarkerResolutionGenerator.JeSSMarkerResolutionIgnore

All Implemented Interfaces:

org.eclipse.ui.IMarkerResolution

Enclosing class:

[MarkerResolutionGenerator](#)

private class **MarkerResolutionGenerator.JeSSMarkerResolutionIgnore**

extends java.lang.Object

implements org.eclipse.ui.IMarkerResolution

The resolution that allows the deletion of a single JeSS marker

Constructor Summary

private	MarkerResolutionGenerator.JeSSMarkerResolutionIgnore ()
---------	---

Method Summary

java.lang.String	getLabel ()
void	run (org.eclipse.core.resources.IMarker marker)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MarkerResolutionGenerator.JeSSMarkerResolutionIgnore

private **MarkerResolutionGenerator.JeSSMarkerResolutionIgnore**()

Method Detail

getLabel

```
public java.lang.String getLabel()
```

Specified by:

getLabel in interface org.eclipse.ui.IMarkerResolution

run

```
public void run(org.eclipse.core.resources.IMarker marker)
```

Specified by:

run in interface org.eclipse.ui.IMarkerResolution

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)
