Assignment

On

Software Inspection Tool

Subject: Software Engineering II

Submitted to: Sir Muzaffar Hameed

Roll#: BSIT07-15

Department of Information Technology
Institute of Computing BZU MULTAN

Introduction and History of Software inspection

M.E. Fagan of IBM first introduced software inspections in 1974. Since that time the technique proposed by Fagan has become well established in many businesses and has resulted in both productivity and quality improvements. Research in the area of software inspection has shown that when used properly, between 50 and 90 percent of the defects in software artifact can be detected shortly after they are introduced. This not only improves the quality of the software product, but dramatically reduces the costs associated with correcting defects because they are uncovered soon after they are introduced.

Unlike walkthroughs and other review techniques, software inspections are formal and follow a well-defined process. The result of each inspection is a formal, quantitative report that contains categorized defect data as well as inspection "process" data such as preparation time. The information collected during an inspection is not only valuable for its positive impact on product quality and costs, but also because it can be used as a process control tool. The software inspection data from one development phase can be used to improve, in "real time" both the current phases and future phases of development. For example, information from an inspection indicating a high number of data definition defects in the design phase may indicate a need for better control of the requirements phase. When this information is "feed back" to the requirements phase, improvements can be immediately be made to the process. On the other hand, information from inspection during the implementation phase that uncovers a high percentage of logic defects could be "feed forward" to improve the testing phase. Software inspection data can also be used to improve the inspection process itself.

List of performed Inspections

- Requirement Inspection
  - DS - Diagnostic System
  - IGUI - Integrated Graphical User Interface
  - DAL - Data Access Library
- Design Inspection
  - TM - Test Manager
  - DS - Diagnostic System
- Code Inspection
  - IPC - Inter Process Communication
  - MRS - Message Reporting System
  - IS - Information Service
  - DAL - Data Access Library
- 180 pages of documents
- 8000 lines of code

Software inspection is a group activity that, as stated earlier, follows a well-defined process. A properly conducted inspection follows the process model shown in Figure 1.

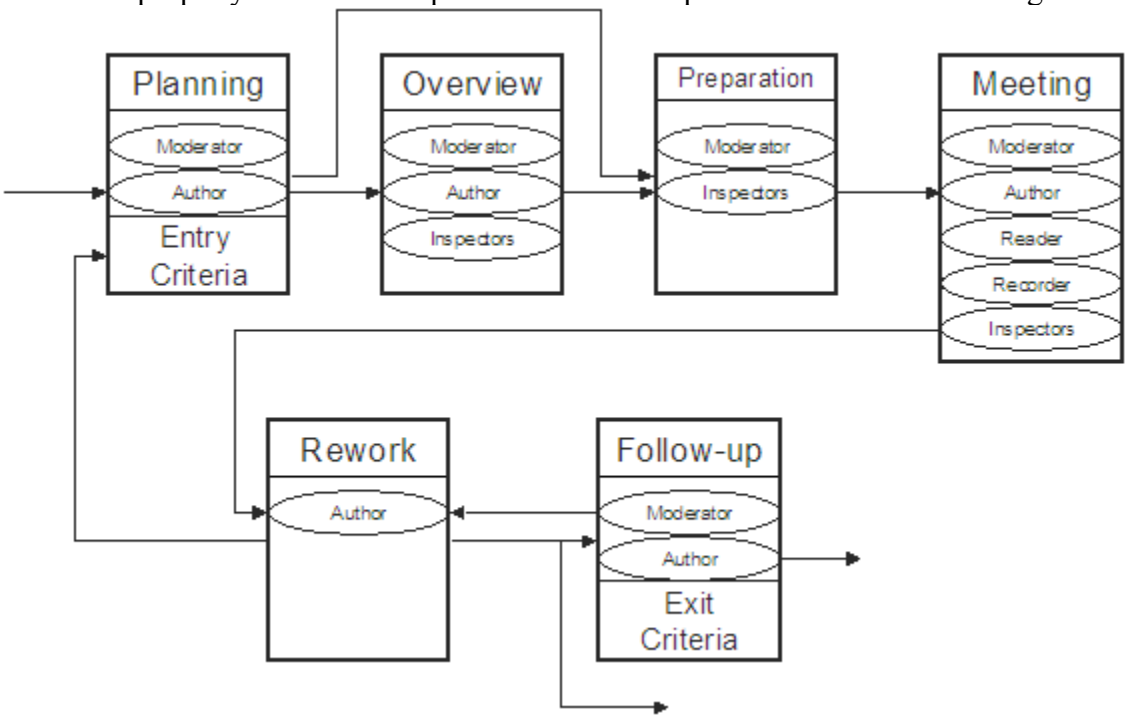


Figure 1: Software Inspection process model (Fagan).

As indicated by the model, there are several roles that should be represented in the various phases of a properly done inspection. They are as follows:

- **Author:** The owner and/or producer of the work product being inspected.
- **Inspector**: A person responsible for detecting defect in the work product, often chosen to represent a stakeholder in the development process (i.e., designer, tester, technical writer).
- **Moderator**: Responsible for organizing, executing, and reporting the inspection.
- **Reader**: Guides the examination of the work product.
- **Recorder**: Enters defect information found during the meeting.

Software Inspection Tools

The task of creating a tool to support automated software inspection is a difficult one. Like system development, the process of software inspection is highly cognitive and the targets of inspection are usually artifacts that lack a high level of formality. The only completely formal software system component that lends itself to any level of automated inspection is the implementation code itself. The informal nature of artifacts such as requirements specification and even design documents make them nearly impossible to inspect these documents "automatically". As the software engineering discipline moves in the direction of formal methods and MDA, automated inspection will likely become more of a reality.

The current selection of tools designed to support software inspection fall into one of two categories. They either support the process by providing functionality that allows collaborative and distributed inspection teams to function effectively or they provide only code inspection capability. This paper will review tools from each category.

Collaborative Inspection Tools

Internet Based Inspection System (IBIS)

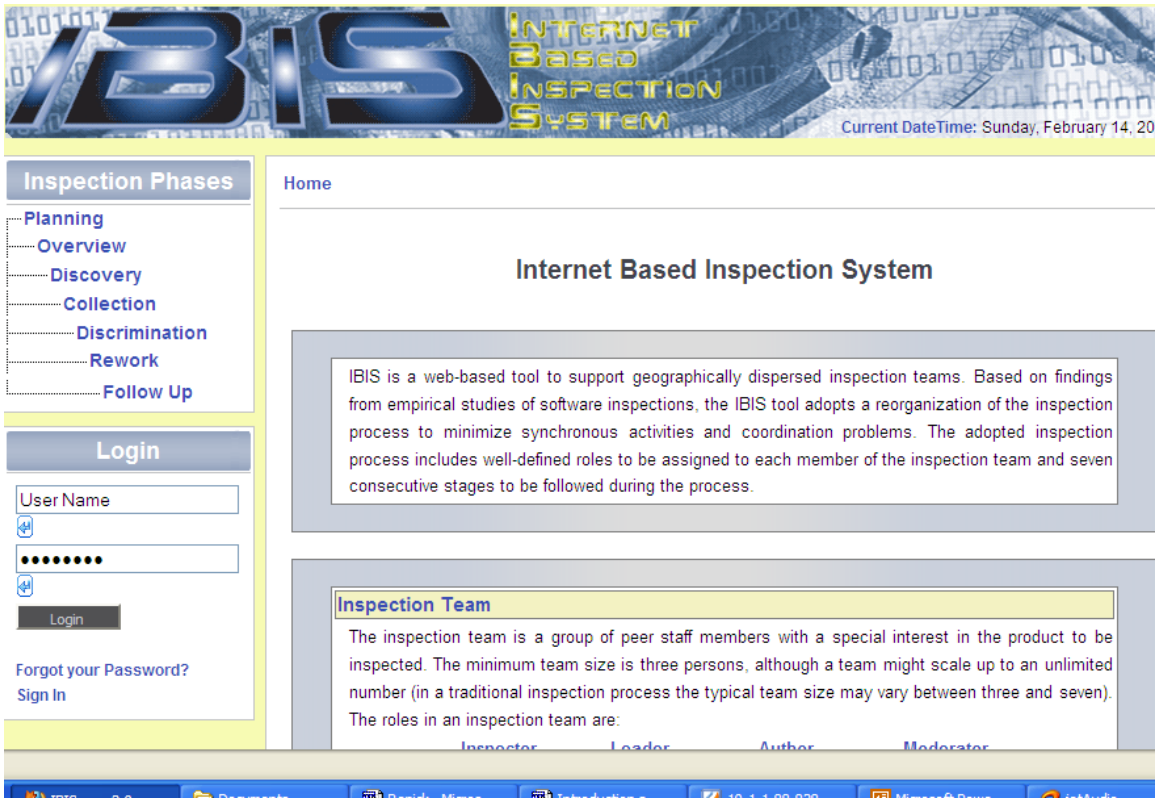


Figure 2: IBIS Version 3.0

IBIS is a tool developed by Lanubile and Mallardo, that supports distributed asynchronous software inspection. The creators noted that there has been a recent history successful distributed open-source projects and chose to use information from these projects as a guide in designing IBIS. Like the typical open-source project, IBIS uses web browsers and email readers as the primary client side communication tools. The use of these common tools increases the probability that a large number of inspectors will participate in a given project and also allows inspections to proceed asynchronously (this is especially useful for geographically diverse teams). A UML deployment diagram describing one view of the IBIS architecture is displayed as Figure 3.

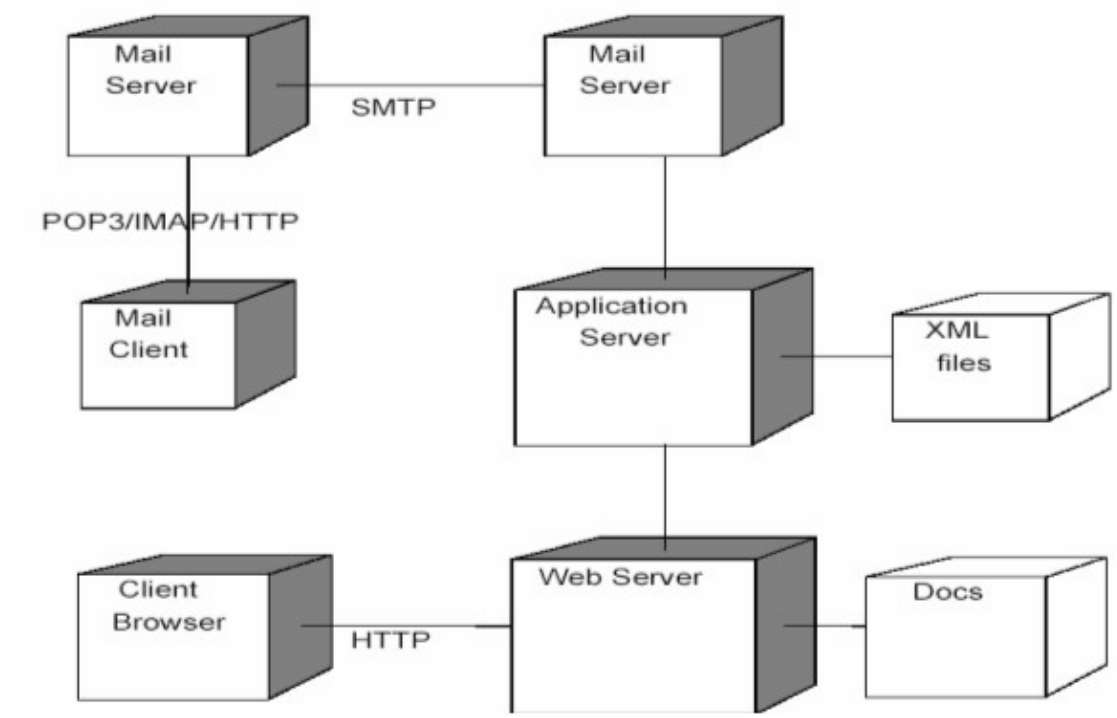


Figure 3: IBIS deployment diagram.

The authors chose to slightly reengineer the inspection process to better support distributed inspection. The reengineered process replaces the inspection meeting with the following steps:

Discovery: Individual examination with the purpose of understanding and defect detection.
Collection: Merge of individual defect lists. Duplicate defects are moved directly to the rework phase.
Discrimination: Small team (or inspector-author pair) examination.

These three steps directly replace the meeting and all other inspection phases remain unchanged.

IBIS provides a template for the planning phase of the inspection. With this template, the moderator can define the artifact to be inspected, the purpose of the inspection as well as the type and severity of the defects to be inspected for. The planning template also provides for entry of each of the inspector's contact information. When the template is complete, IBIS generates e-mail to each of the inspectors notifying them of the details. No special functionality exists for the overview stage outside of allowing the inspectors to access background information.

During the Discovery phase, IBIS provides a log for recording defect information. Only the checklist reading method is directly supported by the system. At any time, an inspector can notify the system that he or she has completed the Discovery phase. When

this notification is received, a message is sent to the moderator and the moderator is then allowed to browse each inspector's discovery log.

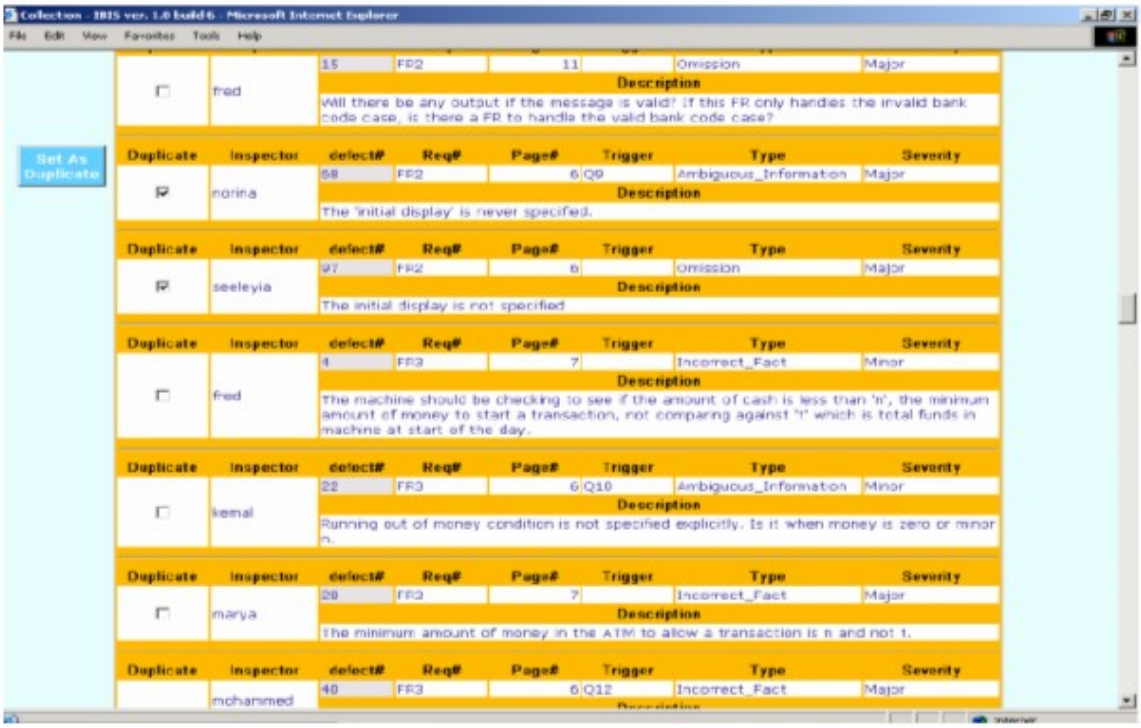


Figure 4: IBIS Merged Inspection Log

The Collection phase commences with each of the discovery logs being merged (see Figure 3). During this phase, the moderator looks for defects reported by more than one inspector (duplicate defects). Duplicate defects are assumed to be valid and moved directly to the rework phase. The search for duplicate defects is simplified by the ability to sort the defect list in various ways.

After duplicate defect have been moved to the Rework phase, the remainder of the defects are moved to the Discrimination stage. Here, the proposed defects are discussed between all of the inspectors through the use of a discussion forum. Each proposed defect is given a unique thread in the forum and the moderator may remove threads when the disposition of a defect becomes evident.

IBIS supports the Rework phase by providing a defect resolution form that must be completed by the author as he or she corrects each defect. As each defect is addressed a notification message to the moderator is generated. The Follow-up stage is supported with automatic generation and distribution of inspection summary reports.

IBIS appears is an inspection tool that does a good job of managing the "clerical" duties surrounding any software inspection. Support for additional reading techniques beyond checklists would make this a much more powerful tool. One of the major advantages of this tool is that it is implemented using common WWW enabled technologies, which should make it easily accessible to virtually any user with a modern PC.

Collaborative Software Inspections (CSI)

CSI was developed by a team of researchers at the University of Minnesota at Minneapolis to support distributed software inspections. The tool is designed to allow the team to use either the Yourdon or the Humphrey inspection techniques, both of which are variants of Fagan's approach. The primary difference between the two techniques and Fagan's is in the preparation stage. Yourdon's version of the preparation process allows inspectors to informally note potential defects and other problems as well as documenting any positive observations regarding the artifact being inspected. In Humphrey's approach to the preparation phase, each inspector develops a potential defect list and forwards it to the work product author before the inspection meeting. The work product author is then expected to address each of the faults during the inspection meeting.

Conduction inspections using the CSI tool require that at least some of the activities be conducted synchronously while others can be done asynchronously. The asynchronous activities that CSI supports are:

1. Distribute target material – planning phase.
2. Review target material – preparation phase.
3. Merge potential faults – preparation phase.
4. Record inspection results – follow-up phase.

Activities that must be completed synchronously if using the CSI system are as follows:

1. Discuss faults – meeting phase.
2. Categorize faults – meeting phase.
3. Determine work product status – meeting phase.

Allowing each inspector to create annotations and creating hyperlinks to the annotation in the source document support the asynchronous activities of the preparation phase. When all of the inspectors have completed their initial review, the moderator is able to merge the annotations into a master list. The master list is then available to all inspectors during future phases of the process. The synchronous inspection activities (primarily the meeting) are supported through real time display of the inspection materials including the work product, fault list, annotation, action item list and a note pad for general observations. Teleconf provides discussion capability. A typical arrangement of windows used during the meeting phase is shown in Figure 5.

Figure 5: Typical arrangement of windows during a CSI based inspection meeting.

The collaborative inspection in CSI is implemented through the use of several TCP/IP enabled components (Figure 6). The Browser component displays the artifacts as well as links to the other components necessary to conduct an effective inspection.

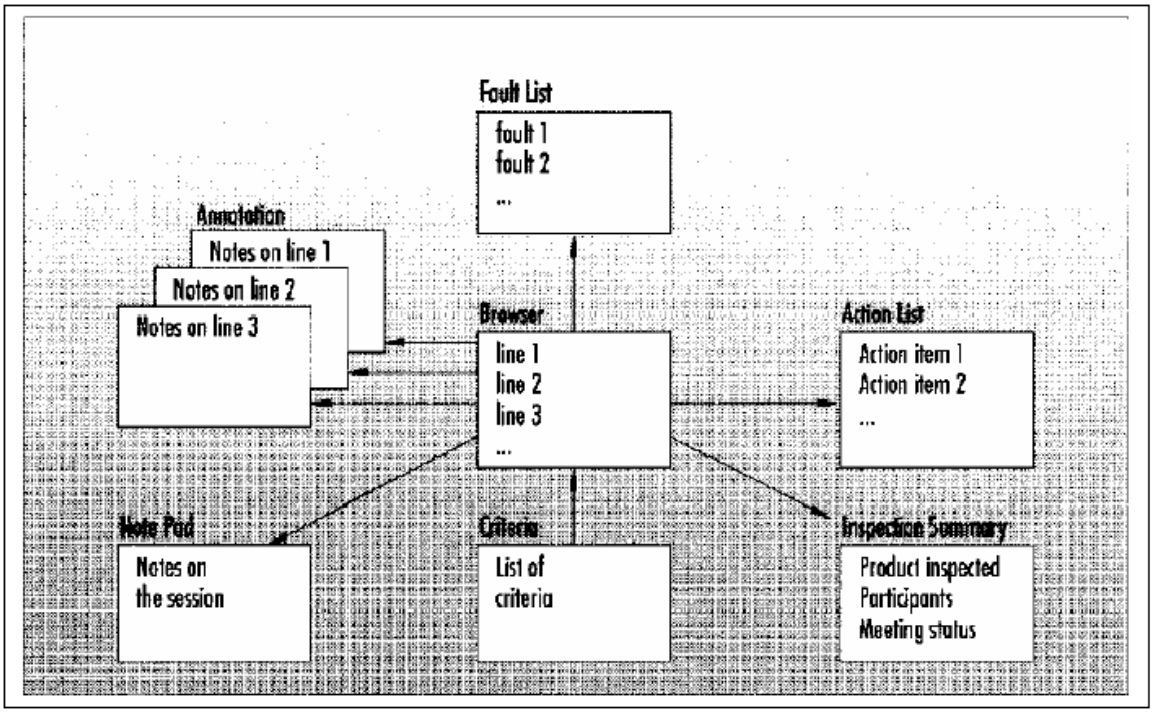


Figure 6: CSI component interaction diagram..

An Annotation component allows inspectors to record defects detected during the preparation phase. The Fault List component is available to merge the individual defect

lists, and to categorize and sort the individual defects for simplified analysis. A Note Pad is available to allow inspectors to record general observations that may apply to the artifact as a whole and not to an individual line of information. The results of the inspection meeting stored in an Action List component that has sorting and categorizing capability that is similar to the fault list. The Criteria component acts as a roadmap for the inspectors, providing them with guidelines for detecting defects. CSI also includes an Inspection Summary component that provides logging and reporting functionality.

When compared to IBIS, CSI is significantly older technology. It appears to be a good general-purpose preparation and meeting tool, but the fact that it is implemented with custom components can present a major drawback. It also appears that CSI lacks the capability to support artifacts that are not text based (like UML diagrams). This presents another potential drawback to today's Software Engineer.

JStyle

JStyle is a Java code inspection tool developed by Man Machine Systems. Its primary function is to parse Java source code and analyze it for common coding problems. JStyle ships with about 100 preinstalled "rules" for evaluating various aspects of the source code. The product also allows the user to create customized rules using either VBScript or JMScript scripting language and providing access to the source code parse tree generated when the code is analyzed. The built in rules fall into several categories and each rule is assigned a severity level between 1 and 7 with one being lowest risk and 7 being highest. Unfortunately, no indication of the meaning of the severity levels could be found in the JStyle literature. In order to present a better idea of JStyle's capabilities, following is an example of one rule from each of the categories:

Category	Rule #	Description	Severity
Class Member Specific	ST1041	Abstract method can't be private or final.	3
Class Specific	ST1044	In a non-public class, there is no need for a public constructor.	3
Exception Handling	ST1009	The 'return' statement in 'finally' block nullifies the effect of 'return' found within 'try' block. Check this design.	7
Finalizer Specific	ST1014	Explicit call to finalize() does not alter the 'gc' state of the object.	5
General	ST1079	The return value of the method call is ignored. Check whether this is intended.	1
Inner/Anonymous Class	ST1045	An inner class of non-public class need not be public.	3
Naming Convention	ST1020	The required interface name prefix is missing.	1
Performance	ST0169	There is no need to make a copy of a String object. Strings are immutable.	4
Redundant	ST1053	This method has a parameter that is not	4

Declaration		used.	
Thread Specific	ST1016	If you catch ThreadDeath, ensure that you throw it back. Otherwise, the thread won't die.	7
Variable Hiding	ST1050	Field in the class hides one of the super class fields.	4

The JStyle user interface is similar to those of common integrated development environments and should be intuitive to most users. It allows easy switching between the source files, comments and metrics. Comments that are generated due to rule violations are hyper-linked to the attributable source for easy evaluation and correction. Code evaluation is completed amazingly fast. JStyle completed analysis of the Azureus project (containing over 850 files, see below) in about 1 minute, making it suitable for use on larger scale projects. A typical display immediately following code evaluation is shown in Figure 7.

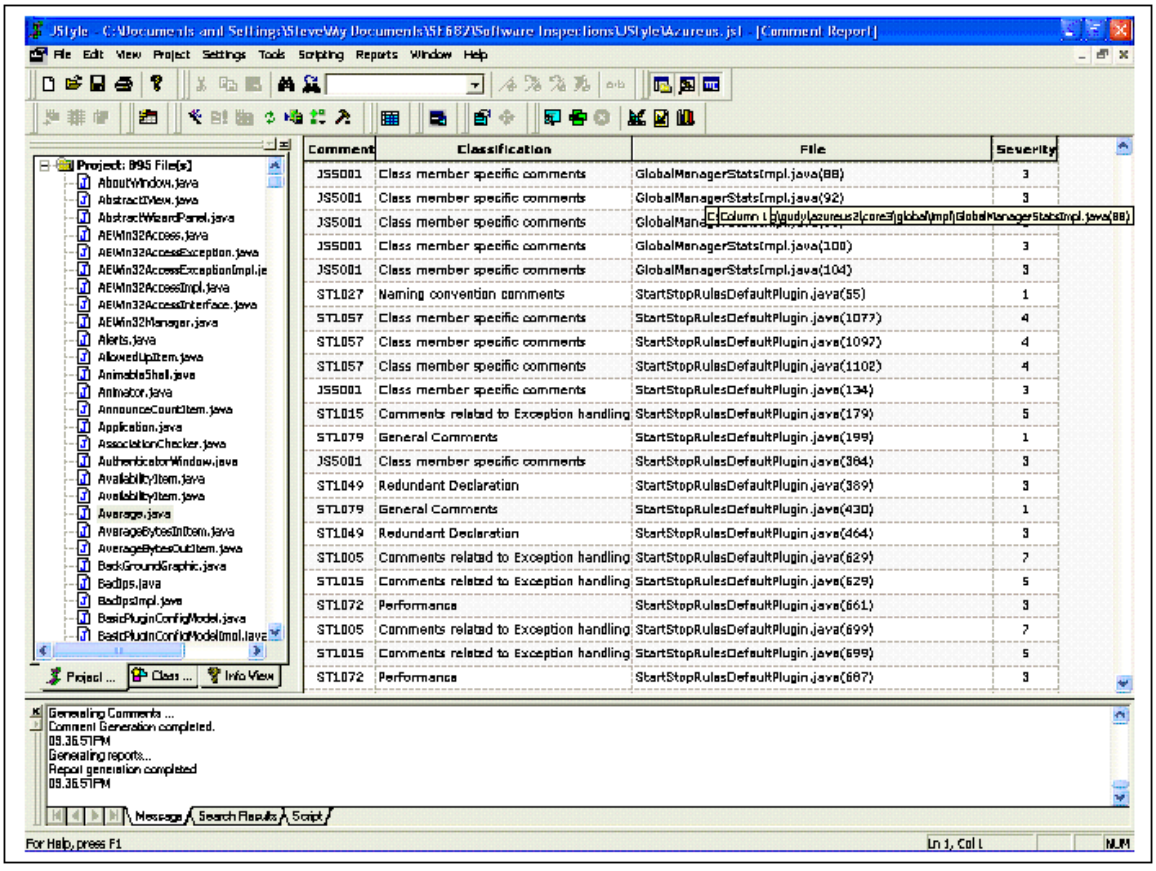


Figure 7: JStyle screenshot immediately following code evaluation.

QStudio

QStudio is a java code analysis tool provided by QA Systems, Inc. The basic version, QStudio Pro, was available for free and was the product that was evaluated for this survey. Like the other tools already discussed, QStudio ships with well over 150 rules for analysis pre-installed. Each of the rules is associated with a quality attribute, a quality sub-attribute, and an impact level. The impact level is a number between 1 and 5 with 1 being the lowest an 5 being the highest which indicates how significantly the rule violation impacts the quality of the Java code. Rules with an impact level of 1 may only represent minor annoyances to the developer while a 5 indicates a violation which could likely lead to product failure. Following are a few selected examples of the rules that ship with QStudio.

No.	Quality Attribute	Quality Sub-Attribute	Rule	Impact
2	Reliability	Failure Liability	Always use method equals() to compare objects.	4
124	Maintainability	Clarity	Avoid shadowing fields of a class or its superclasses and interfaces by local variables of a method.	3
186	Portability	Platform Conformance	Do not use hard-code positions or size of a GUI element.	3
232	Maintainability	Style Conformance	A @return tag is specified for method method name, but this method returns void. Could you remove the @return tag?	2

QStudio also allows the creation of user defined rules. User defined rules are supported through the use of the open source PMD specification which makes use of the Visitor pattern. In short, PMD uses the JavaCC parser generator and JJTree to create an abstract syntax tree form the source code. New rules are added by extending the "net.sourceforge.pmd.AbstractRule" class, and implementing at least one of its visit() methods.

As stated earlier this survey used the "free" version of QStudio. The user interface is simple and intuitive, but this version lacks any reporting or summarizing tools, making it difficult to get an overview of the code inspection results. In addition, code analysis using QStudio takes much longer than with the other tools evaluated. It took about an hour to complete the analysis that the other tools were able to complete in minutes, using the same source files.

AppPerfect

AppPerfect is a Java Development environment that includes many software engineering related tools, one of which is dedicated to code analysis. Like the other tools, AppPerfect ships with numerous "built-in" rules (about 125). The rules are divided into 13 categories and 4 severity levels (low, medium, high, and critical).A few selected rules follow:

Category	Rule	Severity
Optimization	Use BufferedInputStream and BufferedOutputStream or equivalent buffered methods wherever possible; doing I/O a single byte at a time is very slow.	Critical
Portability	System.out.println statements and similar constructs synchronize processing for the duration of disk I/O and can significantly slow throughput.	High
Metrics	Complexity of any method should be less than 6.	Medium
Security	Make classes non-serializable.	Medium

Of the tools evaluate AppPerfect seems to have the most intuitive method for categorizing rules. User defined rules are available through the use pre-defined "general", "function", and "datatype" tags. Unfortunately, the documentation regarding user defined rules was of almost no value in helping understand how to create a customized rule.

The user interface is similar to the other tools that have been discussed and sufficient reporting, view and summarizing features are available. AppPerfect is also capable of calculating several basic project metrics that can be valuable for quantitative process control. Analysis with AppPerfect, while not the quickest of the tools evaluated was still completed in an about 3 minutes.

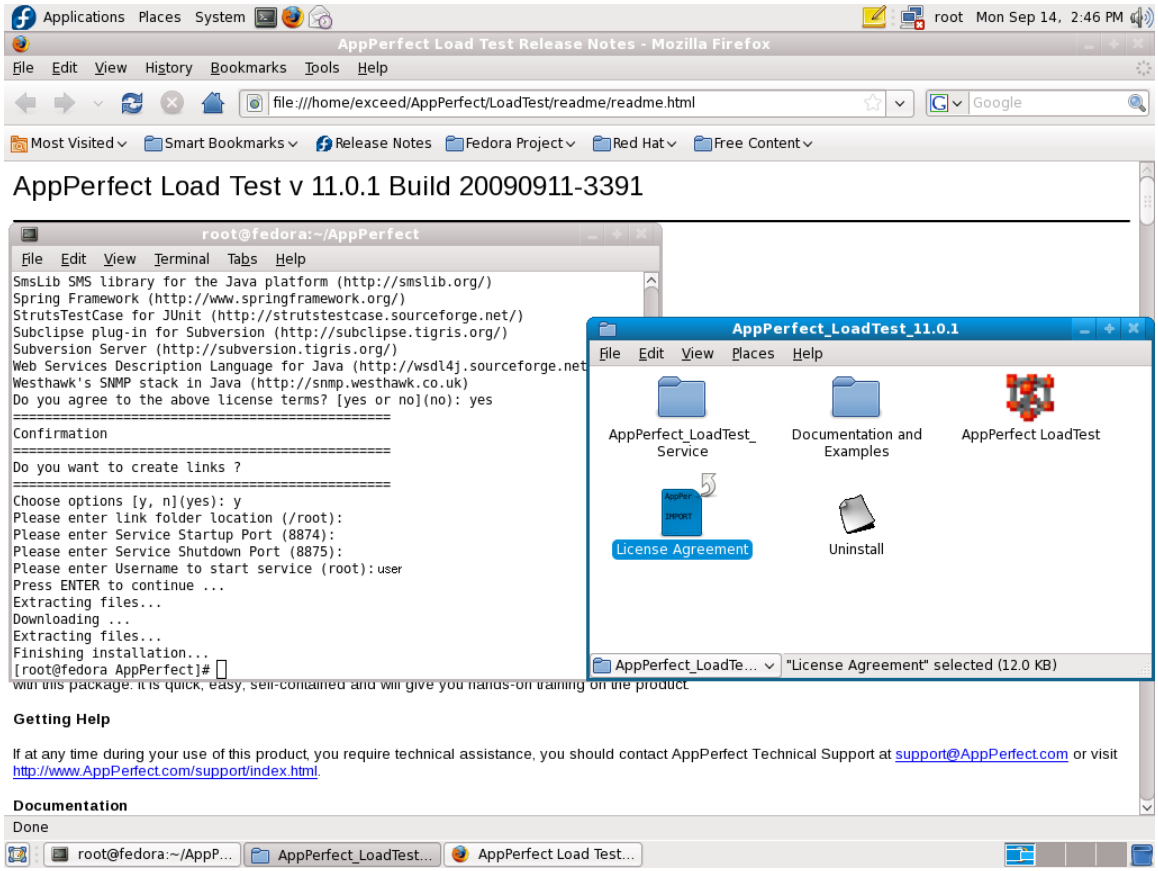
Load Testing: Also referred to as Performance testing or stress testing involves simulating heavy user load to ensure your application or Web site can handle it effectively. AppPerfect experts utilize the AppPerfect Web Load Test product to build sophisticated tests to ensure you can go live with confidence. We can fully automate the testing to make it a part of your process. We can help analyze results and pin-point problem areas.

Functional Testing: Testing software to ensure your applications implement your business objectives. We can also automate these tests to ensure no regressions are introduced over time.

Java Testing: - Java testing services are built around three core product offerings: Java code analysis, Java unit testing and Java profiling. We can automate the Java testing into your development process to make it a seamless means to ensure Java code quality on a

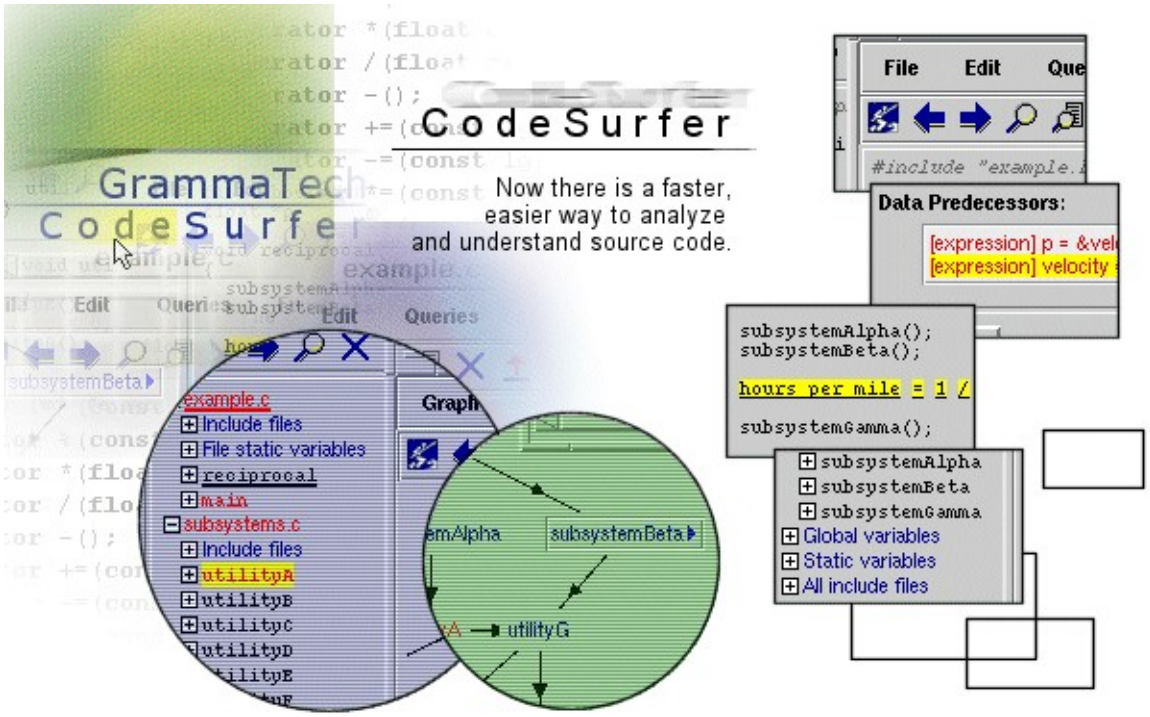
continuous basis. Our services can assist you with complex tasks such as memory leak detection, performance bottlenecks, multi-threading issues, etc.

Product customization: On the rare occasion when our products cannot meet your requirements out-of-the-box, we offer product customization services to add a new feature or modify/enhance an existing feature to meet your needs.



CodeSurfer

Codesurfer is a powerful C and C++ source code analysis and navigation tool. It displays information about your program at an unprecedented level of detail.
Codesurfer is a new tool for easier and more precise navigation and understanding of source code. Codesurfer has many uses, including program understanding, maintenance, impact analysis, debugging, reengineering, and reuse.
Codesurfer is unique because it enables you to identify and navigate the "deep structure" of your program effortlessly. The deep structure comprises the direct and indirect relationships, or dependences, within your source code. These are the semantic threads that reveal exactly how your program works.



JTest by Parasoft

ParaSoft's Jtest 3.0 is a powerful automated tool for testing Java classes. Developers can unit-test their code for completeness and standards compliance and conduct regression tests to ensure that changes they've made to their code haven't introduced errors.

White Box Testing

Anyone who has survived a long testing project knows that one of the most tedious processes is writing test cases. Jtest is the first testing application that generates unit test cases based on the internal structure of your classes. Using patented technology, Jtest examines byte code, trying to break the class by attempting to pass unexpected variables to its methods.

To begin white box testing, open Jtest and browse to the class you'd like to test. To test multiple classes, go to the Project Testing UI and select the directory, zip or jar file of classes. After this is completed, press the start button and wait for Jtest to conduct its tests.

Jlint 3.0

- Jlint check Java code and find bugs, inconsistencies and synchronization problems by doing data flow analysis and building the lock graph.
- Finds unreachable code
- Threading/lock problems
- More than just coding standard checking
- Find bugs that even manual inspections can't find – not even by experienced staff!

PMD

PMD scans Java source code and looks for potential problems like:

- Possible bugs - empty try/catch/finally/switch statements
- Dead code - unused local variables, parameters and private methods
- Suboptimal code - wasteful String/String Buffer usage
- Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
- Duplicate code - copied/pasted code means copied/pasted bugs

Conclusion

- Inspections are better and cheaper in finding defects than testing alone
- Earlier detection of defects are possible by inspections
- Manual inspections do take a lot of time and may not catch all defects for complex multi-threaded OO software
- Static Analysis tools and Reading Techniques alleviate some of these problems
- QA plays a key role in leading the inspection process and educating staff in processes, procedures, static analysis tools and in reading techniques