# Abstract

# Problem Description

Table : - Error Checking

|  |  |  |
| --- | --- | --- |
| Types of Errors | Tools for Essays | Tools for Code |
| Syntactic | Spell Check | Compiler |
| Semantic | Grammar Check |  |
| Substantive | Audience/Grader | User/Tests |

When writing or creating anything from an essay to code, we encounter three different types of errors: syntactic, semantic and substantive. To describe each, let us first examine the process of writing an essay in Word. Word has a Spell Checker that immediately alerts the author that their paper contains a syntactic error – specifically that they have written something outside the bounds of the English language. Word also has a Grammar Checker that reports semantic errors – a passage appears to be poorly written inside the English language. Finally one has an audience reading the essay who will highlight any substantive errors – errors within the arguments or premises. When writing code we have a different set of tools. Compilers immediately alert us of semantic errors – our code is not correct. We also have tests or users that can highly substantive errors – the program is not working correctly. However, the area of semantic error checking – whether code is well written – has yet to be automated fully.

Seen from a different point of view, this problem can be formulated in terms of software quality. Namely, there are two perspectives on software quality: that of the user and that of the programmer. Users care about whether or not a piece of software works (aka behaves as it ought to). In contrast, programmers care about how maintainable a piece of software is. Maintainability minimally implies code is easy to read and update. There are ways to evaluate the user’s perspective of a program, most easily through automated testing. Though there are tools to evaluate the programmer’s perspective, these tools only check specific aspects of code. Given that code quality is subjective, any tool that only performs pre-defined inspections will never be satisfactory to every programmer.

While I have talked about the nature of semantic error checking, I have yet to comment on why it is important. The biggest reason to perform semantic error checking is to improve readability or the ease with which another programmer can understand the code. In the same way that grammatical errors in a paper can confound its underlying arguments, poorly written code can easily obscure its underlying function. In an industry or academic setting where other individuals will necessarily need to read one’s code, readability is at a premium. Similarly, readable code is easy to revise and update later. In an academic environment, automated semantic error checking immediately helps save work for professors and TAs by producing automated comments instead of needing to write the same set of stylistic comments to multiple students. Students can also directly benefit by applying this tool to their code before submitting assignments; giving them the chance to improve their grades and their coding habits.

# Related Products

There are some tools that try to fill this gap with semantic error checking. Each approaches the problem differently but all help to find some semantic errors. The tools that I will be discussing are Splint, PMD, and Checkstyle.

## Splint

Splint is a tool for “statically checking C programs for security vulnerabilities and programming mistakes”. It works exactly as my program will from the student’s perspective, namely as a command-line program which prints warnings and errors to stdout. Without additional information, Splint displays warnings about basic semantic errors like assignments with mismatched types and ignored return values. With some more effort, programmers can add annotations (effectively fancy comments) that give Splint a specification to check against. These annotations allow for stronger checks like memory management, null pointers and “violations of information hiding” (p. 9). Examples of annotations in action are the following abstract type declarations:

Figure : - Splint Annotations

typedef /\*@abstract@\*/ /\*@mutable@\*/ char \*mstring;

typedef /\*@abstract@\*/ /\*@immutable@\*/ int weekDay;

These annotations define mstring and weekDay as abstract types and further specify that they are mutable/immutable respectively.

While these annotations provide an extensive feature set, they are a hugely inconvenient. They require programmers specifically write their code to meet the specification, not only of the client, but also of the tool. For new programmers (often the ones who need the most error checking), these annotations are almost impossible to implement on top of learning to program. As described by Evans in an email, “One of the goals of the original design of Splint was for programmers who add no annotations to start getting some useful warnings right away, including warnings that encourage them to start adding annotations.  For some aspects, such as /\*@null@\*/ annotations I think this has worked okay, but for others like abstract types, memory management, etc. I don't think it has worked very well, and the warnings on these issues tend to either make developers want to stop using Splint, or at least just turn off all the warnings of that type, rather than start adding the annotations needed to enable better checking.”

## PMD and Checkstyle

PMD is a tool for checking Java Code that is integrated into a dozen or so different popular IDEs. PMD comes premade with over 250 checks broken up mostly by purpose such as Braces Rules, Basic Rules, Coupling Rules, etc. Some of these checks also deal explicitly with a certain library or platform like Android, Jakarta and JUnit. PMD works by passing source code into a JavaCC-generated parser and receiving an Abstract Syntax Tree (AST, basically an object containing a model of the source code). PMD then traverses the AST and calls each rule in the RuleSet to check for any violations. This pattern of examining a tree of nodes is called the Visitor Pattern. The RuleSet is an XML file that can be edited to augment the functionality of PMD with customized rules. Rules are written in their own classes and extend a base implementation. The rule itself can override three functions (start, visit and end) to perform various checks against the source code based on the nodes in the AST. Here is the “dummy” example from the PMD website which counts how many expressions are in the source code:

Figure : - PMD Example Rule

package net.sourceforge.pmd.rules;

import java.util.concurrent.atomic.AtomicLong;

import net.sourceforge.pmd.AbstractJavaRule;

import net.sourceforge.pmd.RuleContext;

import net.sourceforge.pmd.ast.ASTExpression;

public class CountRule extends AbstractJavaRule {

private static final String COUNT = "count";

@Override

public void start(RuleContext ctx) {

ctx.setAttribute(COUNT, new AtomicLong());

super.start(ctx);

}

@Override

public Object visit(ASTExpression node, Object data) {

// How many Expression nodes are there in all files parsed! I must know!

RuleContext ctx = (RuleContext)data;

AtomicLong total = (AtomicLong)ctx.getAttribute(COUNT);

total.incrementAndGet();

return super.visit(node, data);

}

@Override

public void end(RuleContext ctx) {

AtomicLong total = (AtomicLong)ctx.getAttribute(COUNT);

addViolation(ctx, null, new Object[] { total });

ctx.removeAttribute(COUNT);

super.start(ctx);

}

}

Checkstyle mimics PMD in function but provides different checks out of the box (namely those regarding duplicate code, class design, whitespace, etc). It also uses the Visitor Pattern and provides ways to examine the beginning and ends of the tree as well as each node. New rules are added through an XML file and similarly passed to Checkstyle at runtime.

## Plugin Architecture

### Eclipse

### YouTube