Japroch: A Tool for Checking Programming Style

S. Mäkelä and V. Leppänen
University of Turku and TUCS, Department of Information Technology, Lemminkisenkatu 14A, FIN-20520 Turku, Finland

sajuma@utu.fi, ville.leppanen@it.utu.fi

Abstract

Following a good programming style is essential basis for making programs. To aid the teaching of Java programming, we present a general style checking tool that visually shows the places in programs, where a chosen style is not followed. Our tool has a configuration language that enables one to describe many kind of good coding conventions.

1 Introduction

Many studies, Mengel and Yerramilli (1999); Oman and Cook (1990); Rees (1982); Truong et al. (2004), emphasize that when teaching programming for the first year students, it is very important to express, what good programming style means. Almost every programmer has a vague understanding on good programming style, but finding a rigid definition is difficult. A notable exception in this respect is 'Java Code Conventions', Sun Microsystems (2004), although it can be argued whether the conventions capture all aspects of good programming style and whether all the values for style parameters are the best possible. To some extent the matter is subjective. However, studies of Berry and Meekings (1985); Mengel and Yerramilli (1999); Miara et al. (1983); Sun Microsystems (2004); Oman and Cook (1990); Rees (1982); Truong et al. (2004) list a similar set of attributes for defining a good programming style. The proposed ideal 'values' for attributes differ though. The idea in many studies has been to develop a metrics to measure the quality of programming style.

In this paper, we describe an interactive tool, *Japroch* (Java programmable checker; see Figure 1), that can be used to check whether a Java 5.0 program is written according to some style. Because there is no consensus on a single good programming style, our tool is very flexible supporting many kinds of style checking aspects via its configuration language. The current version of the tool focuses on typographical issues, as do Miara et al. (1983) and Oman and Cook (1990), but the tool can also reveal some violations related to semantical or logical usage of constructions. Moreover, there is a clear idea (Section 4), how our tool can be generalized to support checking more demanding properties of programs, related to the semantics of certain constructions. Besides providing feedback for the students, one purpose of this tool (as in Jackson (1996)) is to decrease the time spent on checking student programming exercises.

1.1 Related work

Some of the early studies Berry and Meekings (1985); Kernighan and Plauger (1982); Leach (1995); Miara et al. (1983); Rees (1982) involved implementing a checking tool for Pascal or C programs. Some of those studies even considered Cobol and Fortran. Newer studies consider Ada (and C; Jackson (1996)), C++ (Mengel and Yerramilli (1999)), and Java (Truong et al. (2004)).

The tool described in Truong et al. (2004) focuses on finding syntax errors, semantic errors, and logical errors. Some of the errors can be considered as style errors, but mostly their tool operates in a different domain than ours. Mengel and Yerramilli (1999) do not really construct a style analysis tool, but use Verilog Logiscope tool to extract style related parameters from programs. Their main idea is to form a style metrics for C++ programs. The Ada related tool reported in Jackson (1996) considers correctness, efficiency, complexity, and test data coverage besides the style issues. In the style issues, the tool follows guidelines given by Berry

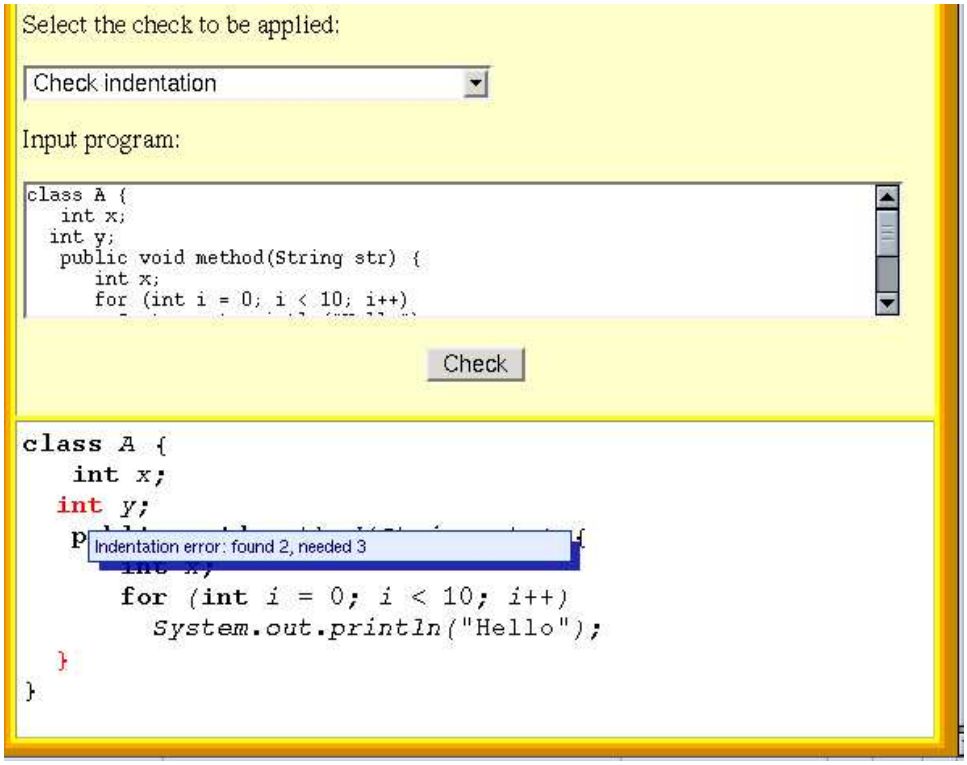

Figure 1: Screenshot of the user interface. The tool is instructed to use a very simple style 'check indentation', which only looks for indentation errors (found errors shown in red).

and Meekings (1985). The tool reports numerical values for all the aspects – those could be used to define a metrics for Ada programs. The configurability of the Ada tool is unknown to us.

In Internet distribution, there appears to be one extensive tool, Checkstyle, Burn (2003), for checking the style of Java programs. The tool is formed to match the Java coding conventions provided by Sun Microsystems (2004). Checkstyle has XML-based configuration file, where the user essentially can specify, which predefined checks are applied and with which parameter values (e.g. maximum length of lines, value for indentation). The tool does not seem to allow flexibly construct new kind of checks as our tool does. On the other hand, Checkstyle is currently more extensive than our tool. Checkstyle reports found style errors by giving line numbers – whereas our tool shows them visually. Reporting of Checkstyle can be made more visual by configuring it to work with Eclipse.

2 Elements of good programming style

What are the elements of good programming style? Space limitations prevent us to try to fully answer to the question. Perhaps, the most fundamental work is Oman and Cook (1990), where a taxonomy for good programming style is defined. Their style taxonomy consist of four parts: general programming practices, typographical style, control structure style, and information structure style. With respect to object-oriented languages, the last part seems obsolete (the taxonomy is written with respect to C and Pascal).

Practical studies of Berry and Meekings (1985), Jackson (1996), Kernighan and Plauger (1982), Leach (1995), Sun Microsystems (2004), Miara et al. (1983), Mengel and Yerramilli (1999), Rees (1982), and Truong et al. (2004) do not fully follow the proposed taxonomy, but rather list style topics that are considered. Also the Checkstyle tool, Burn (2003), does not use the taxonomy to classify its features.

We consider that style issues should be classified into four categories:

Typographical. All issues that are related to the visual outlook of the program. These are mainly related to the usage of white space characters. E.g. indentation, placement of parenthesis, maximum length of line, and method length (in number of statements).

Syntactic. All issues, where some usage of language construction is syntactically correct, but represents bad programming style. E.g. every switch-statement should have a default-branch, and each case-branch should end to a break-statement.

Semantic. This refers to cases, where the semantics of a construction affects the way it should be used. E.g. class names begin with a capital letter and each declared variable should be used in the program.

Logical. Issues related to the logical structure of the program. E.g. there should not be too deeply nested loops, methods should not have a huge number of parameters, and global variables should not be used as method parameters.

Currently our tool supports typographical and syntactic categories well. Some issues belonging to the other two categories can also be checked, but a more extensive support requires modifications to the tool (see Section 4). The recent study of Halonen et al. (2004) presents a tool for checking the semantic style issues of Symbian C++ coding conventions.

3 Style checking tool for Java

We have implemented a tool that can check for several style related issues in Java programs. New checks can be easily implemented using our configuration language. Currently the checker is about 3000 lines of code, and the libraries for parsing and printing are about 20000 lines.

The program is used through a web interface. With it the user can input a program and select which kind of check will be applied to the program. In the web page generated by the program, we have the locations with errors colorized, and when the user moves the mouse over these locations, we show the related error messages, see Figure 1.

3.1 Features

The styles that we check for are implemented as configuration files. The format of these files will be described later. Currently the program can check for simple typographical issues like indentation and whitespaces, and syntactic checks like checking that all switch-statements have default-cases, or checking that some elements are in a desired order.

There are several different ways to indent the program. For example the placement of braces might be different. Our program can enforce various indentation styles, and many more can be defined easily. The tool can also check for whitespaces in for-, if-, do-, and while-statements.

There are different possibilities on how to indent local variable declarations:

Custom indentation:
```
int x = 0,
    y = 23;
```

Default indentation:
```
int x = 0,
   y = 23;
```

Because code editors have automatic indentation, but they cannot be configured for custom format, it is probably advisable style to forbid declaring several different variables in the same statement. We have also implemented this check.

It is a common error to use operator = instead of ==. Because of this problem, we implemented a check that checks that assignment operators are not included inside other expressions. Even if their uses would be correct, those can make the program harder to read.

We have also implemented checks that ensure that some elements are in a desired order. These include checking modifier order, checking that variables are declared in the beginning of a block, and checking that members of a class are defined in a specified order.

Some other checks are checking for default-case in switch-statements, checking that all cases in a switch are terminated by break, checking for length of lines and checking for redundant boolean literals. It is also possible to check for C-style array declarations in variable and method declarations.

3.2 Configuring the checker

To check for style we need to traverse the parse tree. The inner nodes of the parse tree are syntactic elements of the Java 5.0 programming language. They have a list of subnodes. The leaf nodes are string tokens.

The tree is traversed using the following architecture: A traversing component gets a tree node and an environment as arguments. The component can access the tree, but cannot modify it. The component can access and modify the environment. As a result, checking of the component returns a success, a failure, or a partial failure which means that there was a style error. In addition to these components for nodes, there are similar components for lists of tree nodes. These components are like normal components, but they additionally have a pointer to an element of the list. This enables iterating the elements of the list.

There are several ways to combine components to implement more complex checkers. Boolean operators **and**, **or** and **not** make checks that require both checks to pass, one of the checks to pass or the check to not pass. Iteration structure $*\,c$ applies check c to the elements in the list until all elements are checked or it finds a node that fails the check. Option structure $?\,c$ applies check c to the element in the list, and if succeeds, moves to the next item in the list.

Selection structure **guard**$\{c_1 : a_1 \ldots c_n : a_n\}$ applies checks $c_1 \ldots c_n$ to the node in order. If a check c_i succeeds, to the corresponding action a_i is applied to the token. If none of the checks succeeds, the first partially failed case is selected. The action can add a warning to the checked node, or it can apply more checks to the node.

Figure 2 has an example of the use of the **guard**-construct. It is usually used to traverse the tree recursively, with cases that detect special kinds of elements. The checker defined in the example is given name *root*. It means that it is applied to the top node of the the parse tree. The case **class** *LabeledStatement* checks if the node is of class `LabeledStatement` in the internal representation. When a labeled statement is found, the program outputs a warning. For leaves of the tree there is no check. If the node is an inner node (ie. not leaf), all of its subnodes are traversed recursively to find all labeled statements.

root = **guard**{
  class *LabeledStatement* : **warn** "Labeled statement";
  leaf : **true**; **true** : [$*$ *root*];
}

Figure 2: Example use of the guard-construct.

The environment contains indentation levels in a stack and error messages that are associated with nodes of the parse tree. If a check in the selection structure fails, the changes it made to the environment must be cancelled. Because of this the old environment must be stored. This implies much copying. To make copying efficient we implemented the environment using functional data structures.

For indentation and whitespaces, we can check if the token is indented properly, check if the token has correct number of leading or following whitespaces, and start or stop new indentation level.

Indentation levels are stored as a stack in the environment. To handle various possible styles, there are many different kinds of indentation levels, e.g. levels relative to the first token that belong to the level, or levels that are relative to the number of nested braces. There is

also a feature that checks a leaf token against a regular expression. This can be used to check that an identifier has a standard prefix or postfix.

Because the components can be combined easily, it is simple to write a configuration file parser that can be used to implement checkers.

4 Discussion

Currently our program supports mainly checking syntactic features. An exciting direction for future research is incorporating semantic checks to our tool. To implement this, we need to associate into the tree nodes type information and perhaps some additional information collected from an overall analysis of the program classes (and of the stardard library classes). That information we could use by introducing a new action into the configuration language that would allow calling arbitrary tree nodes checkers written in Java (using the reflection). Such checkers could be implemented using the design pattern 'command'.

We still need to consider the expressiveness of the current configuration language. For example, it should be expressive enough to express almost all of the Java Code Conventions Sun Microsystems (2004), but this has not been verified yet.

Finally, it should be noted that so far we have not made an extensive experimental evaluation of the usefulness of our tool, but once the discussed extension to the tool is made, the evaluation should take place. At that point, we should also construct definition(s) for good programming style taught at our university.

References

Berry, R., Meekings, B., 1985. A Style Analysis of C Programs. Communications of the ACM 28 (1), 80–88.

Burn, O., 2003. Checkstyle Home Page. Http://checkstyle.sourceforge.net/.

Halonen, H., Närvänen, K., Mikkonen, T., 2004. Automatic Checking of Symbian Coding Conventions. In: Proceedings, 11th Nordic Workshop on Programming and Software Development Tools and Techniques NWPER'2004. pp. 79 – 88.

Jackson, D., 1996. A Software System for Grading Student Computer Programs. Computers & Education 27 (3/4), 171 – 180.

Kernighan, B., Plauger, P., 1982. The Elements of Programming Style. McGraw-Hill, Inc.

Leach, R., 1995. Using Metrics to Evaluate Student Programs. SIGCSE Bulletin 27 (2), 41–43.

Mengel, S., Yerramilli, V., 1999. A Case Study of the Static Analysis of the Quality of Novice Student Programs. In: Proceedings of 30th SIGCSE Technical Symposium on Computer Science Education. ACM Press, pp. 78–82.

Miara, R., Musselman, J., Navarro, J., Shneiderman, B., 1983. Program Indentation and Comprehensibility. Communications of the ACM 26 (11), 861–867.

Oman, P., Cook, C., 1990. A Taxonomy for Programming Style. In: Proceedings of the 1990 ACM Annual Conference on Cooperation. ACM Press, pp. 244–250.

Rees, M., 1982. Automatic Assessment Aids for Pascal Programs. SIGPLAN Notices 17 (10), 33–42.

Sun Microsystems, 2004. Java Code Conventions. Http://java.sun.com/docs/codeconv/.

Truong, N., Roe, P., Bancroft, P., 2004. Static Analysis of Students' Java Programs. In: Proceedings of Sixth Conference on Australian Computing Education. Australian Computer Society, pp. 317–325.