JPP: A Java Pre-Processor

Joseph R. Kiniry and Elaine Cheong
Caltech Technical Report CS-TR-98-15
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

September, 1998

Abstract

The Java Pre-Processor, or JPP for short, is a parsing pre-processor for the Java programming language. Unlike its namesake (the C/C++ Pre-Processor, cpp), JPP provides functionality above and beyond simple textual substitution. JPP's capabilities include code beautification, code standard conformance checking, class and interface specification and testing, and documentation generation.

1 Introduction.

The Java Pre-Processor, referred to as JPP henceforth, is a parsing pre-processor for the Java 1.X programming language. This document will describe the design, development, and use of JPP.

JPP is primarily used as either a front-end for a Java compiler like *javac*, *espresso*, or *guavac*, or as a verification and validation tool as part of a configuration management system. We will describe both possibilities here as well as provide a number of examples of innovative alternative uses.

The default JPP configuration corresponds to the Infosphere's Java Coding Standard[1], a freely available code standard developed at Caltech by the Infospheres Research Group[2]). Any questions as to the details of code standards, layout, beautification, and verification are addressed in that document. This document only focuses on the JPP tool.

All aspects of JPP are user configurable. Configuration is accomplished through the use of Java properties and command-line switches.

1.1 JPP Functionality Summary

Briefly, JPP provides the following functionality:

[1]http://www.infospheres.caltech.edu/resources/code_standards/java_standard.html
[2]http://www.infospheres.caltech.edu/

- *Code Beautification.* JPP can process any legal Java code and reorganize it in an ordered, aesthetic manner. The rules used in the reformatting of code are user configurable. JPP is used as a code beautifier to clean up downloaded or otherwise adopted code before performing a code review. It is also used in preparation for gaining an understanding of a codebase before modifying and maintaining it. Finally, JPP can be used to process code from multiple developers/teams regularly to enforce local coding standards.

- *Code Evaluation.* JPP can also be used to take an existing piece of code and "grade" it. Such a code evaluation comes in several forms:

  - *Standard Conformance Checking.* JPP is configured to a specific code standard. Code standards give rules on how code is syntactically arranged, variable naming policies, feature access rules, etc. JPP can evaluate how well a piece of code conforms to its current code standard. Such an evaluation comes in the form of a short or long report which can be automatically mailed to the appropriate parties and/or logged for review.
  - *Code Complexity Analysis.* JPP knows several standard code complexity algorithms. JPP can evaluate Java classes, interfaces, and even whole packages and provide complexity measures. These metrics can help guide a designer and developer toward more readable, understandable, and maintainable code. Metrics are also provided as short or long reports.
  - *Documentation Analysis.* JPP can evaluate the thoroughness of code documentation. JPP knows what a "real" comment is and performs information theoretical complexity measures (per-feature and per-entity entropy) of comments to determine their completeness and usefulness. JPP will comment on your comments!
  - *Object-Oriented Design Principles.* JPP understands a core set of object-oriented design principles: the Laws of Demeter, class, interface, and inheritance dependency graph analysis, JavaBean patterns, and component specification to name a few. JPP can evaluate Java code and provide suggestions as to how to improve design based upon these principles.

- *Class and Interface Specification.* JPP's specification constructs are inspired by Meyer's *Design by Contract.* The primary constructs used to specify a contract for a component are method preconditions and postconditions, and class invariants. JPP can enforce and/or encourage the specification of contracts on the methods of classes and interfaces, and can test for the validity of such clauses in subtyping relationships.

- *Class and Interface Testing.* In addition to the aforementioned contract specifications on methods, JPP supports the specification of loop invariants and variants. All five of these constructs (the three specification clauses and the two loop clauses) can be transformed by JPP into actual embedded test harness code and inserted into a class's methods at the appropriate points. The insertion of the test code at compile time, and the enabling and disabling of test code at run-time, is completely under the control of the developer.

- *Documentation Generation.* JPP can also transform Java code into HTML, XML, or LaTeX for direct code documentation and reference. The resulting document is properly formatted for pretty-printing and has the appropriate embedded links for feature cross-referencing. External language-specific links can be embedded in the code with special directives to the pre-processor.

JPP is an evolving tool. If you have suggestions or bug reports, please email the authors at `jpp@unity.cs.caltech.edu`.

2 Using the JPP

This section describes how to configure and use JPP. Note that even without configuring JPP at all it will perform quite well for most users.

2.1 Configuration

JPP is configured with the use of Java *properties*. Properties are either specified with property files, with Java environmental variables, or some other VM-specific mechanism.

2.1.1 Example Property File

A Java property file is simply a list of property value pairs. A default property file can be created by JPP by using the "`write_defaults`" switch (see below). For example, a legal property file would be the following (from the JDK 1.2beta4 distribution):

```
#  @(#)flavormap.properties        1.4 98/03/03 1.4, 03/03/98

#
# This properties file is loaded by java.awt.dnd.FlavorMap class
# on loading and contains the Motif/X11 platform specific default
# mappings between "common" X11 Selection "target" atoms and a
# "platform" independent MIME type string.
#
# It is required that there is 1-to-1 (inverse) mapping between
# platform targets and MIME strings.
#
# these "defaults" may be augmented by specifying the:
#
```

```
#  awt.DnD.flavorMapFileURL
#
# property in the appropriate awt.properties file
# this will cause this properties URL to also be loaded into the
# FlavorMap.

STRING=text/plain; charset=iso8859-1
FILE_NAME=application/x-java-file-list;class=java.util.List
```

In this property file, two properties are defined, `STRING` and `FILE_NAME`. `STRING` is given the value `text/plain; charset=iso8859-1`, and `FILE_NAME` has the value `application/x-java-file-list;class=java.util.List`.

2.1.2 Setting a Property: An Example

For example, suppose that the property `ADDRESS` should be set to the email address of the user running JPP. For this author, the proper address would be `kiniry@cs.caltech.edu`.

Property Files. On most UNIXes and Windows, this would be accomplished by adding the line

```
# Setting the property ADDRESS to the string "kiniry@cs.caltech.edu".
ADDRESS=kiniry@cs.caltech.edu
```

to my property file. Note that all lines that start with a "#" are comments and are ignored. We will discuss the JPP-specific contents and location of property files later in this document.

2.1.3 Location of JPP Properties File(s).

JPP searches for the property files for a project in the following places and order:

- In the user's home directory. More specifically, whatever the Java runtime thinks the user's home directory is.
- In the directory specified with the *-project* command-line option (see below).
- In the current directory.

Properties specified in these three locations are cumulative/additive. Later specifications override earlier ones.

Java Property Variables. Alternatively, a property can be set with a Java property variable. Java property variables are set by using the `-D` switch with the `java` command. Consult your local documentation for more information.

2.1.4 Command-Line Options

Command line options are an alternative way of configuring JPP. As mentioned previously, command-line options take precedence over environmental variables and property files.

JPP is not completely configurable from the command-line. Some of JPP's functionality is too complex to specify on the command line. Additionally, given the breadth of capability of this tool and the finite length of most command lines due to the limitation of many shells, attempting to set every option on the command-line seems foolhardy.

The following command-line options are available:

-beautify
: Perform code beautification.

-standard_conformance (warning | fail)
: Check conformance to a code standard and choose what action to take upon conformance failure. The default option is *warning*.

-complexity (ALG_k)*+*
: Analyze code complexity with one or more algorithms. The algorithms should be provided as a comma-separated list. For example, `-complexity alg1,alg2,alg3`. No spaces should be included in the list.

-documentation (warning | fail)
: Analyze thoroughness of source documentation and, upon conformance failure, what action to take. The default option is *warning*.

-design (ALG_k)*+*
: Evaluate code with one or more object-oriented design principles.

-verify_specification (warning | fail)
: Verify class and interface specification and, upon conformance failure, what action to take. The default option is *warning*.

-assertion (none | pre | post | invariant | loop | check | all)
: Perform class and interface testing to varying degrees:

- `none`: no assertions are checked
- `pre`: only preconditions are checked
- `post`: only postconditions are checked
- `invariant`: only class invariants are checked
- `loop`: loop invariants and variants are checked
- `check`: check instructions are executed
- `all`: everything is checked

-generate_documentation (HTML | XML | LaTeX)
: Generate code as documentation in one of several formats.

-write_defaults
: Write the JPP default settings for all options to a property file in the local directory, which the user can then modify as they see fit.

-f
: Use JPP as a filter. Input should be directed to *stdin*. JPP's output will be sent to *stdout*, and error messages will be sent to *stderr*.

-o (filename | prefix)
: Specify output filename or file prefix. Generated files, depending upon which options are set, are either sent directly to the file specified or to files starting with the provided prefix. The prefix is used to prefix all output files (e.g. `prefix.html`, `prefix.report`, etc.).

-k
: Specify that JPP should continue to operate for as long as possible even in the presence of errors (like make -k).

-help
: For help on JPP's command-line options.

-project (directory)
: The location of the main project directory.

-debug
: If you want to see debugging output.

-source_extension (extension)
: Sets the extension of the original source code. I.e. If your source uses the extension "`.j`", then you should pass "`j`" to this parameter. This extension defaults to "`j`".

-destination_extension (extension)
: Sets the extension of the destination source code. This extension defaults to "`java`".

2.1.5 Command-line Properties and Property Files

Properties can be set either in a property file or via a Java property option, (as mentioned previously, usually `-D`), on the command-line. If a property is specified in both the command-line and in the property file, the command-line option takes precedence.

2.2 Usage

Normally, JPP is used from the command-line as a filter to pre-process Java source code for a Java compiler. Because most Java compilers will not compile source provided via a pipe, in general a two-stage mechanism is necessary.

However, a two-stage mechanism is problematic in Java because of the naming restrictions placed upon Java source files.

We have provided several alternate mechanisms for running JPP with the standard JavaSoft Java compiler. The most simple mechanism is a simple shell or batch script similar to the following:

```
#!/bin/sh
jpp -f < $$@ > $$.java)
if [ $? == 0 ];
then
  javac $$.java
fi
```

We suggest naming Java source files with an alternative suffix such as "`.j`" or "`.jav`". This way, JPP can process the source file and generate a proper "`.java`" file that a compiler will accept.

2.3 Makefiles

Activating JPP should be switchable via your development environment. For the standard Unix *make* tool, we suggest using the following makefile rules as templates. "`ClassName`" is just a placeholder for your own class name.

```
ClassName.java: ClassName.j
        jpp ClassName.j

ClassName.class: ClassName.java
        javac -g -deprecation ClassName.java
```

If you are using GNU make, this can be (significantly) shortened to the two following implicit rules:

```
%.java: %.j
        jpp $<

%.class: %.java
        javac -g -deprecation $<
```

These rules will insure that each source file (suffixed with "`.j`") is translated by jpp into the corresponding "`.java`" file that the Java compiler can handle.

2.4 Project Version Control

JPP-processed source files should not be added to a version control system. Similar to object files ("`.o`", "`.class`", etc.), processed files contain no additional information and thus would only waste space in a repository.

3 Design of the JPP

This section discusses the analysis and design stages of the development of JPP. During the analysis stage, we perform a requirements analysis and determine the objectives of the project. During the design stage, we decide how to fulfill these goals.

3.1 Analysis

During the analysis phase, we first determine the tool requirements and the reasons and background for these requirements. We will then determine the top-level means by which we can accomplish these goals and investigate what impact these choices will have on development. We also develop a common ontology — our project dictionary — during the analysis phase so that everyone working on the project (and reading about it afterwards) has a common vocabulary.

3.1.1 Tool Requirements

In designing the Java Pre-Processor (JPP), we first analyzed our requirements for the tool. These requirements were introduced in Section 1. Here, we will discuss them in further detail.

Code Beautification. It is very easy to create ugly code. Poor layout, confusing indentation, and non-standard coding styles can make program code very difficult to understand. A tool that takes compilable but incomprehensible code and generate "pretty" code is called a pretty-printer.

One of JPP functions is that of a pretty-printer for the Java programming language. The tool should recognize the Java language and the user should be able to specify the layout of the code output, including indentation, location of braces, commenting style, etc.

Code Evaluation. JPP should also enforce good software engineering methodologies by providing algorithms to check conformance with a code standard, analyze code complexity, check documentation thoroughness, and evaluate adherence with a variety of object-oriented design principles.

Every programmer has her own style of programming. However, in any organization, it is often beneficial to specify a standard *coding style*. This code standard often includes rules on documentation, variable naming, class and variable typing, and code layout. See [7] for a comprehensive example Java coding standard.

One of the requirements imposed on JPP includes the enforcement of local code standards. Such standards should be user-specifiable and completely flexible. JPP should ignore, generate warning messages, or produce a report describing non-conforming code. The first version of JPP should implement the Infospheres Java Coding Standard and only be customizable with respect to syntax, not documentation semantics.

Another way to help developers create easy-to-understand code is to provide a tool that analyzes code complexity. JPP should provide this functionality and allow the user to choose among several different complexity analysis algorithms.

Once a programmer has created simple, "clean" code that matches local coding standards, she can improve its usefulness by thoroughly documenting the code. JPP should aid in the development of thorough code documentation via a documentation analysis function. This aspect of the tool should measure the amount of actual code and "real" comments and provide such information to the programmer via complexity reports.

As an object-oriented language, Java code should be written with object-oriented design principles in mind. JPP should help programmers improve the overall quality of their code design by providing a function to check the code for conformance to various design methodologies. JPP should allow developers to analyze their code using the Laws of Demeter, class, interface, and inheritance dependency graphs, JavaBean patterns, component specification patterns, and more.

Documentation Generation. We should also design JPP to automatically generate documentation. Similar to Javadoc, JPP should use standard tags enclosed inside of special documentation comment blocks identified with the strings `/** ... */`. Developers should also be able to use JPP with "pluggable" subcomponents for alternative documentation type generation, e.g. XML, LaTeX, and roff. Thus, JPP should be extendable so that the generation of documentation in other personalized or proprietary formats is possible.

Class and Interface Specification and Testing. First, we provide the reader with background on Design by Contract and assertions. Then, we discuss the details of what JPP should provide to help the user implement these concepts through the specification and testing of Java classes and interfaces.

3.1.2 Specification and Testing Fundamentals

Background. *Design by Contract* [12] is a software engineering methodology created by Bertrand Meyer, author of the Eiffel [13] programming language. *Contracts* between the suppliers and clients, in a software context, are written by associating a specification with every software element[4]. This contract promotes a better understanding of software construction, provides an effective framework for software reliability and quality assurance, and leads to more effective and complete documentation software components.

Assertions. Design by Contract's core construct is that of the assertion. An assertion is a predicate which states a logical sentence that evaluates to *true* or *false*. The assertion is embedded in program code and, if during program execution the assertion evaluates to false, an error is indicated. There are three main types of assertions:

precondition - a condition that must be true at the beginning of a section of code, usually a method.

postcondition - a condition that must be true at the end of a section of code, again, usually a method.

invariant - a condition that must be true at all *stable* points in program execution. We will discuss issues of *stability* later in this document.

Invariants. There are several types of invariants. A *class invariant* is an assertion describing a property that holds for all instances of a class and, potentially, for all static calls to the class. Two other types include *loop invariants* and *loop variants.* A *loop invariant* is an assertion that is true at the beginning of the loop and after each execution of the loop body. A *loop variant* is an assertion that describes how the data in the loop condition is changed by the loop. Loop variants are used to check forward progress in the execution of loops (i.e. to avoid infinite loops and other incorrect loop behavior).

While invariants specify predicates which remain true, a program is a discrete system, and thus invariants are often temporarily violated. In general, for object-oriented systems, the following rule holds: a public (class or method) invariant specifies a predicate which holds true at the instant program execution enters and leaves a public method body. Only at these points in the execution trace is the program state considered *stable* and can thus be tested.

Class and Interface Specification. JPP should enforce the specification of contracts on the methods of classes and interfaces by requiring these clauses to appear in user-specified documentation sections of the code. JPP should also encourage the use of class and interface specifications by generating warning messages when they (the specifications) do not appear. JPP should also check the validity of the contracts, especially when code (class) inheritance is used.

Aside: There is a distinction between specification of contract and full specification of semantics. For contract specification, you can only specify truths that are externally visible to the client object. With full specification, you can detail (abstractly) (semi-)complete behavior and/or semantics. The first version of JPP should only allow the user to establish contract specifications. Future versions might implement the full specification of the semantics of classes.

Class and Interface Testing. C and C++ compilers come with libraries (e.g. `assert.h`) that let developers to specify assertions in the code. Java does not come with this capability.

We should design JPP such that it allows the programmer to check assertions by using special tags (e.g. Infospheres Java Coding Standard special Javadoc tags) to include expressions for preconditions, postconditions, and invariants in the comments.

The developer should be able to use the tool to automatically insert expression-testing code into the original code. This is an extremely useful program testing and debugging feature. Command-line switches or some other preference configuration feature should be used to enable or disable the test code at runtime. Thus, these specifications serve a dual purpose: that of providing code documentation and a test suite harness for assertion checking.

3.1.3 Necessary Algorithms

JPP needs to know several algorithms for full functionality:

- object-oriented code complexity metrics [1, 17, 2]
- documentation analysis metrics ($LOC_{omments}$, Lines of Comments, information theoretical complexity: per-feature and per-entity documentation entropy, etc.)
- object-oriented design principles (Laws of Demeter[11]; class, interface, inheritance, dependency graph analysis[5]; JavaBean "pattern" conformance; component specification completeness)

3.1.4 Fulfilling the Tool Requirements

The tool should understand Java so that it can manipulate the code correctly. It should be able to read, understand, and output compilable Java code. The user should also be able to select which features of the tool she would like to use each time, so that the tool performs different functions depending on what is specified for each particular run of the tool.

3.2 Design

In this section, we describe the details of what JPP will do. First, we discuss why the tool should be designed as a parsing pre-processor. Then, we look at the various aspects of assertion checking. Finally, we specify the tool interface and performance goals.

3.2.1 Creating a Pre-Processor

We would like a tool that can be used in all design and implementation environments, including different brands of compilers, different platforms, and different editors or shells. It is logical to use the Java programming language to create a tool that will manipulate Java code for a variety of platforms.

From our requirements analysis, we determined that we would like to use JPP to do several kinds of code transformations. One possibile way of accomplishing this would be to use Perl or another language that can process text files and manipulate the code, perhaps by using regular expressions. However, the easiest way to accomplish this task would be to parse the code directly. Building a parser gives us the capability to extend the parser and provide more functionality based upon this framework.

3.2.2 Visitor Patterns

The parser can create or store an abstract syntax tree (AST) as it parses the Java source code. An AST is a data structure that, in this case, represents the structure of the parsed code, including keywords, braces, beginning and end of methods, etc.

Once an AST is built, a *tree walker* or *visitor* can be used to traverse and manipulate the tree to create the desired output. We will design several different visitor patterns:

- a "pretty" code pattern for printing,
- an assertion transformation pattern, and
- several documentation generation patterns

3.2.3 Code Evaluation

JPP will evaluate code from several perspectives. The three primary facets of code evaluation are code standard conformance, documentation coverage, and design analysis.

Code Standard Conformance. JPP will enforce user-specified coding standards. JPP will parse a code and compare its syntax and structure with that of user-specified syntax and structure rules. JPP will generate reports to highlight locations of non-conformant code and suggest alternatives. These reports can come in the form of text documents, HTML, or even Email.

The first version of JPP will only implement the Infospheres Java Coding Standard. JPP will ignore or generate an error message for an unknown or incorrect comment tag. Extension to user-defined code standards is under development.

Documentation Coverage Our research group has metrics for documentation coverage as part of both our Java coding standard as well as existing components that check such conformance. Example rules include (a) every feature has a legitimate comment, (b) every Javadoc comment block is complete (i.e. completely documents all aspects of the feature being commented), and (c) the $LOC_{omments}/LOC_{ode}$ (Lines of true comments divided by lines of true code) ratio must be at least 40%.

In other words, our internal metric is that every piece of code must be at least 40% comments before we release it. This ratio will be user-adjustable in JPP.

Design Analysis Design analysis comes in two forms: code complexity metrics and design analysis.

JPP knows several standard code complexity metrics algorithms. JPP can evaluate Java classes, interfaces, and even whole packages and provide reports of complexity measures. These metrics can help guide a designer and developer toward more readable, understandable, and maintainable code. Metrics are also provided as short or long reports.

Additionally, JPP has a set of "design rules" which it can check. Examples of such rules can be found in [7].

3.2.4 Assertion Checking

We wish to support the specification of preconditions, postconditions, class invariants, loop invariants, and invariants. JPP will generate test code using the Infospheres Debug package (called IDebug [8]) to insert assertions into the original program code to check the code against the specification.

The user will specify assertions using special tags inside of documentation comments, which will appear before each method and class declaration that he wishes to test.

The following code block shows how the user can specify assertions inside of a documentation comment.

```
/**
 * @precondition (Expression) <Throwable> Description.
 * @postcondition (Expression) <Throwable> Description.
 * @invariant (Expression) <Throwable> Description.
 **/
```

JPP will parse and recognize `@precondition`, `@postcondition`, and `@invariant` tags. It will store the specified expressions in a data structure for later use. Then, after JPP has parsed the entire program, it should use the parse tree to determine where to insert the assertion testing code.

The expressions specified after any `@precondition`, `@postcondition`, and `@invariant` tags will be inserted into program code at the correct places in the following manner:

`@precondition` This tag should appear in the documentation comment before each method. The accompanying test expression is inserted at the beginning of the method, before any statements of the method are executed.

`@postcondition` This tag should appear in the documentation comment before each method. The accompanying test expression is inserted at the end of the method and before each return statement that exits the method.

`@invariant` This class invariant tag should appear in the documentation comment at the beginning of the class. The accompanying test expression is inserted at the beginning and end of each method in the class.

If multiple `@precondition`, `@postcondition`, and `@invariant` tags are used, the accompanying expressions will be combined conjunctively (per method or class). These are also added to the data structure mentioned previously, which stores the specified expressions.

If any assertion fails, the specified `Throwable` will be constructed and thrown. The text of the assertion specification will be passed to the constructor of the Throwable so that proper context is provided to the debugger and/or developer.

Debug Package. The Infosphere's Debugging package is an advanced debugging framework for Java. This package provides standard core debugging and specification constructs such as assertions, debug levels, stack traces, and specialized exceptions. We will use this package with JPP to implement the assertion checking when inserting test code that corresponds to the assertion tags into the original code.

Inheritance of Assertions. Assertions specified for the methods of one class should still hold true for for inherited methods. JPP supports the standard DBC weakening of preconditions and strengthening of postconditions.

3.2.5 Tool interface

JPP can be run on the command-line. Use of command-line switches and Java properties were described in Section 2.1.

JPP will create one directory inside of the directory that contains the source files, in which it will store temporary files, data files, documentation generation, etc. The output directory can also be specified on the command line. There will be several temporary files per class within the project.

JPP will process the Java source files (recursively, if there are additional subdirectories), rename them with the `.java` extension, and place them in the same directory as the source file. So, Java source files should not be named with the `.java` extension, since they will be overwritten.

Now that we have discussed the design of the JPP, our choice of creating a parsing pre-processor, various visitor patterns, plans for assertion checking, and the tool interface, we will proceed to the a discussion of the actual implementation of the tool.

4 Implementation of the JPP

In this section, we will first introduce the reader to lexers and parsers. Then, we will discuss details of the implementation of JPP.

4.1 A Short Introduction to Lexing and Parsing

JPP uses a lexer and parser to analyze Java source code. To understand how JPP works, we first familiarize the reader with these terms and other related concepts.

Lexical analysis (or *scanning*) is the first stage in processing a language, usually for compilation. The source program is fed into a *lexer* (also known as a *scanner*) as a stream of characters. The lexer groups these characters into *lexemes* (or *tokens*), which are word-like elements, such as keywords, identifiers, and punctuation. These elements are indivisible units of the language.

Next, the *parser*, (or *recognizer*), processes the stream of tokens and determines whether the syntactic structure matches its *grammar*. A grammar is the formal definition of the syntactic structure of a language. The parser can also build an *abstract syntax tree* (AST) as part of its output. This data structure contains the parser's internal representation of the parsed code.

The first parsers were handwritten. Today, there are a variety of parser generators available for a wide range of languages. A parser generator takes a grammar (which could be specified in BNF (Backus Naur Form) or another type of syntax specification) and outputs source code for a parser. Once compiled, the generated parser will recognize valid statements and expressions and perform associated actions.

There are several varieties of parsers that use different algorithms for recognizing valid code structure. The two kinds of parsers that we used in the development of JPP were: (1) DFA-based parsers and (2) LL(k)-based parsers.

DFA Parsers. DFA stands for Deterministic Finite-state Automaton. This finite state machine takes an input event and the current state and uses a state transition function to determine the next state and the appropriate output event. It is deterministic because a single input event uniquely determines the next state. A DFA-based parser is intuitive; each parsed token advances the DFA to the next state until the appropriate terminal state is reached. For example, a semicolon signals the end of a statement.

LL(k) Parsers. An LL parser scans from *l*eft to right using *l*eftmost derivation. LL parsers can parse input without backtracking. With leftmost derivation, the parser replaces the leftmost nonterminal symbol with the matching definition of a grammatical rule. It repeats this process until all nonterminal symbols are replaced by terminal symbols. LL(k) parsers require k tokens of lookahead to decide which rule to apply from a given grammar.

4.2 JavaCC and ANTLR

To create a working pre-processor, we need to be able to parse the code, recognize the Javadoc tags, and insert assertions at the appropriate places in the code. We experimented with two Java parser generators: JavaCC [16] and ANTLR [14]. Both JavaCC and ANTLR come with sample grammars for parsing Java 1.1 code.

Both tools let a developer include arbitrary Java code blocks in the grammar. Such code can be used to scan or parse complex expressions, manipulate data structures, etc. We started implementing JPP with JavaCC, but eventually switched to using ANTLR because of problems with the tool.

4.2.1 JavaCC

JavaCC (Java Compiler Compiler) is a Java parser generator written in Java. First developed at Sun Microsystems, the Java Compiler Compiler (formerly known as Jack) project began as an effort to build a Java parser for QuickTest (now known as JavaSpec), SunTest's Java API testing tool. The Java Compiler Compiler has evolved into a complete set of tools, including JavaCC, JJTree, JJDoc, and JavaScope[3]. We used JavaCC and JJTree.

JavaCC is a DFA-based parser generator. It builds recursive-descent parsers. A recursive-descent parser is a top-down parser built from a set of mutually recursive procedures, each of which implements a grammar production rule. The structure of the resulting program closely mirrors the grammar from which it was generated. Hand-built parsers are usually recursive-descent parsers.

A JavaCC grammar is specified using code-like extended BNF. Both the lexical and grammar specification are contained in the same file. JavaCC also provides lexical state and lexical action capabilities (such as `TOKEN`, `MORE`, and `SKIP`). There is also a `SPECIAL_TOKEN` feature that enables the programmer to define special tokens (such as comments) to ignore during parsing.

JavaCC comes with JJTree, a pre-processor for JavaCC that inserts parse tree building actions at the appropriate places in the JavaCC grammar source. The design of JJTree is based on PGen [15], a tree building parser generator designed by a group at Stanford University. The output of JJTree is run through JavaCC to create the parser. By default, JJTree generates code to build parse tree nodes for each nonterminal in the language. Although JavaCC is a top-down parser, JJTree constructs the parse tree from the bottom up.

For more information, see the JavaCC website[4].

4.2.2 ANTLR

ANTLR [14] (Another Tool for Language Recognition) is a parser generator that can build LL(k) parsers implemented in Java or C++. Originally called YUCC, ANTLR was created as part of the PCCTS (Purdue Compiler Construction Tool Set[5]). This project began as a parser-generator project for a graduate course at Purdue University. PCCTS eventually evolved to include three tools: ANTLR, DLG, and SORCERER. At first, ANTLR included only a parser generator. DLG (DFA-based lexical-analyzer generator) was a lex-like lexical analyzer generator. SORCERER was a grammar-specified tree-parser generator.

Today, these three tools have been combined into one. ANTLR 2.x.x accepts three types of grammar specifications: parsers, lexers, and tree-parsers (also

[3]See the MetaMata home page and the SunTest home page for more information.
[4]http://www.metamata.com/JavaCC/
[5]http://www.ANTLR.org/pccts133.html

called tree-walkers). The current version of ANTLR at the time of this writing (2.4.0) has been completely rewritten in Java.

For more information, see the ANTLR home page[6].

4.3 Development with JavaCC

We started with JavaCC v0.7.1 and later tried 0.8pre1. First, we obtained the sample JavaCC Java grammar (`Java1.1.jj`). The original grammar specified three styles of comments:

```
SINGLE_LINE_COMMENT     //
FORMAL_COMMENT          /** ... */
MULTI_LINE_COMMENT      /* ... */
```

Next, we modified the definition of FORMAL_COMMENT to try to recognize Javadoc tags within the comment.

We attempted to tokenize the tags, but a variety of problems with lookahead surfaced. We also tried to use UNICODE to specify ranges of characters that should be parsed as natural language, but these definitions conflicted with other definitions of the Java language in the grammar. These problems, in addition to outdated documentation, sparsely commented example code, and very poor error reporting made us consider switching to ANTLR.

4.4 Development with ANTLR

We used ANTLR v2.3.0 and started with the accompanying sample Java grammar `java.g`[7]. ANTLR supports *grammar inheritance*, thus we can extend the existing Java 1.1 grammar with our own new productions, adding and overriding the base grammar.

Grammar Extensions. Several extensions to the base Java 1.1 grammar are necessary to accomodate our new constructs, especially with respect to embedded, structured, semantically meaningful comments.

The Java 1.1 grammar comes with 2 styles of comments; one for single line comments and one for multi-line comments.

```
SL_COMMENT              //
ML_COMMENT              /* ... */
```

We added a new comment type called `DOC_COMMENT` (for Java DOCumentation COMMENT). To recognize this new comment type, we needed to differentiate between "normal" multi-line comments (those that begin with the string "`/*`") and documentation comments (those that begin with the string "`/**`").

We first overrode ML_COMMENT (multi-line comment) so that it would not accept documentation comments:

[6] http://www.antlr.org/

[7] We later moved to ANTLR 2.4.0.

```
ML_COMMENT :
  "/*"
  ( '\n' { newline(); }
    | ~('*'|'\n') )
  ( { LA(2) != '/' }? '*'
    | '\n' { newline(); }
    | ~('*'|'\n') )*
  "*/"
{
  $setType(Token.SKIP);
  System.err.println("ML_COMMENT:");
  System.err.println($getText);
}
;
```

Next, we added definitions for documentation comments:

```
// Javadoc documentation comments.
DOC_COMMENT :
  "/**"
  ( { LA(2) != '/' }? '*'
    | '\n' { newline(); }
    | (JAVADOC) => JAVADOC
    | ~('*'|'\n') )*
  "*/"
{
  System.err.println("DOC_COMMENT:");
  System.err.println($getText);
}
;
```

New Tokens A token was added for each new Javadoc tag introduced by the code standard. Example tags include:

```
// metainfo
protected AT_AUTHOR : "@author" ;
protected AT_HISTORY : "@history" ;
```

Then, we needed to be able to recognize natural language, (the descriptions that are normally part of documentation comments), and paranthesized expressions. A production rule was added for each.

```
protected NATURAL_LANGUAGE returns [String natural_language]
{natural_language = null;} :
  ~(' '|'\t'|'\n')
  (~'\n')*  {natural_language = new String($getText);}
  '\n' {newline();}
;

protected DOC_COMMENT_EXPRESSION
```

```
  returns [StringBuffer doc_comment_expression]
{doc_comment_expression = null;} :
  '('
  ( DOC_COMMENT_EXPRESSION
    | ~')'
  )*
  ')'
{
  doc_comment_expression = new StringBuffer($getText);
}
;
```

Finally, we added productions to recognize documentation comment tags. Here are two examples of such productions, one for the simple `@author` tag, and one for the more complex `@invariant` tag.

```
// Bug tag
// @bug Description of the bug.
protected BUG returns [String bug]
{ bug = null; } :
  (' '|'\t')+
  bug = NATURAL_LANGUAGE
  { System.err.println("Bug tag: " + bug); }
;

// Invariant tag
// @invariant (Expression) Description.
protected INVARIANT returns [String invariant]
{
  invariant = null;
  StringBuffer doc_comment_expression;
} :
  ({LA(1) != '('}? ~'(')*
  doc_comment_expression = DOC_COMMENT_EXPRESSION
    {
      System.err.println("DOC_COMMENT_EXPRESSION: " +
        doc_comment_expression);
      if (invariant_data.length() != 0) {
        invariant_data.insert(0, '(').append("&&").
          append(doc_comment_expression.toString()).append(')');
      } else {
        invariant_data.append(doc_comment_expression.toString());
      }
      System.err.println("@invariant_data = " +
        invariant_data.toString());
    }
  (' '|'\t')+ invariant = NATURAL_LANGUAGE
     {System.err.println("invariant tag: " + invariant); }
;
```

Production Modifications. We specified an initial required documentation comment in the parser compilation unit:

```
// Compilation Unit: In Java, this is a single file.  This is the
// start rule for this parser
compilationUnit :
  // A compilation unit starts with an optional package definition
  ( packageDefinition
    | /* nothing */
  )

  // Next we have a series of zero or more import statements
  ( importDefinition )*

  // A documentation comment must come next.
  ( documentationComment )+

  // Wrapping things up with any number of class or interface
  // definitions
  ( typeDefinition )*

  EOF
;
```

4.4.1 Current State of JPP

JPP has the following assertion-checking functionality:

- Creates a `jpp_data` directory if it does not exist.
- Saves preconditions, postconditions, and invariants to a file in the `jpp_data` directory.
- Saves method signatures to a file in the `jpp_data` directory.

ANTLR also has tree building capabilities. The default tree created in the parser after activating the tree building option (`buildAST`) is a linked list of tokens (a degenerate AST). By enabling this option, we can generate the AST and use a visitor to print out parsed code, transform it for code beautification, etc.

4.5 Using JavaCC/ANTLR

When developing the Java Pre-Processor, we started with JavaCC and eventually switched to ANTLR. By using both of these lexer/parser/tree parser generators, we discovered various advantages and disadvantages that made it easier or harder to create the JPP tool.

We started with JavaCC, since this particular package seemed to be the most popular Java parser generator available. Support included a FAQ, a mailing list (with archive) and a newsgroup. Initially, it seemed like a good package to use, since there were many example grammars included with the distribution, including a very useful Java 1.1 grammar. However, as we started actually implementing the JPP, we found that it was diffcult to modify the grammar to include support for recognition of Javadoc tags. We did not understand the JavaCC syntax very well, and the JavaCC documentation does not explain these details very well. Major problems with the documentation included references to variable names that were not updated with the recent implementation of JavaCC. Also, debugging the JPP grammar was very difficult, since the generated error messages were usually not very helpful. Overall, it was easy to start using JavaCC (in that the grammars were easy to understand), but development soon became extremely difficult when our new comment specification conflicted with other parts of the grammar (we were unable to detect where they conflicted because of the poor error reporting).

We soon switched to ANTLR, having come across this package during our initial search for Java parser generators. There was also an ANTLR newsgroup, but a mailing list was only just recently started (as of this writing). ANTLR did not come with as many example files as JavaCC, but they adequately demonstrated the various ANTLR features. Many of them were similar to the applications included with JavaCC, which made the transition to ANTLR very easy. This also came with the all-important Java 1.1 grammar. There was plenty of on-line documentation available for nearly all aspects of the ANTLR package. A reported bug list was also very helpful when we came across an inconsistency in the generated Java code. ANTLR error reporting was much more accurate and detailed than the messages generated by JavaCC.

We found that the most useful features of the ANTLR package were: grammar inheritance (for extension of the Java 1.1 grammar) and `protected` grammar rules (easy to make subrules for Javadoc tags). The use of rule parameters and token variables enabled the passing of values between productions rules and saving this information to a data structure for later use.

4.6 Optimization

Until we use JPP in several large projects we will not know what kinds of optimization, if any, are necessary to increase JPP's performance. In general, we would like the tool to be as non-intrusive as possible, especially with respect to compilation time. We will supply data on performance when it becomes available.

5 Conclusion

The Java Pre-Processor is a comprehensive package that can assist Java programmers with many aspects of the development process. Initially, it can be used to evaluate the design according to object-oriented design principles. The developer can also use JPP to check contracts on class methods and interfaces. Checking assertions with JPP can help shorten the time-consuming debugging phase. The programmer can further improve his code by using JPP to check for code complexity. Additionally, JPP can be used to beautify code and ensure that it conforms to local coding standards so that other developers can read and understand the carefully crafted code. Finally, JPP can automatically generate software documentation in various formats from the documentation within the code. Even this can be evaluated by JPP for completeness and usefulness. By using JPP for each stage of development and improving the Java code based on JPP evaluations, the software engineer is well on his way to creating well-structured, easy-to-read, well-documented, reusable, and bug-free code.

5.1 Other Work

Several packages provide or describe functionality similar to various subcomponents of JPP including iContract, source beautifiers, Lint-like tools, JavaNCCS, and JML.

iContract. iContract by Reto Kramer[9] is a freely available source-code preprocessor which handles class invariants, pre- and post-conditions. It also uses special comment tags (e.g. @pre, @post) which are converted into assertion checks code that is inserted into the source-code. The expressions are a superset of Java, compatible with a subset of the latest UML Object Constraint Language (OCL)[6]. Highlighted features include old- and return-value references in post-conditions, implications and the naming of exception classes to throw. iContract fully supports the propagation of invariants, pre- and post-conditions via inheritance and multiple interface implementation, as well as multiple interface extension mechanisms. The instrumentation level (e.g. only pre-condition checks) can be chosen on a per file level enabling fine grained, vertical (inheritance, implementation) and horizontal (delegation) performance control throughout your system.

Thus, iContract has an advantage on the existing version of JPP because it has extra specification constructs derived from OCL. We are working in this direction as well. All other iContract functionality is provided by JPP.

Source Beautifiers. The C Beautifier, often called `cb` on many UNIX systems, is an example of a tool that transmutes source code (in this case, the C language) into a more regular, structured, "beautiful" format. Tools that have equivalent functionality include `cc-mode` in Emacs, and the grind toolsuite (`cgrind`, `vgrind`, etc.). JPP has a Java beautifier built in that inherits much of its configurability from Emacs' `cc-mode`. Other tools that fall into this category include UNIX's `enscript` and GNU `indent`.

Lint-like Tools. `lint` might be considered the original source code verifier. Lint-like tools check source code for a variety of "bad-practice" violations (improperly initialized variables, unused code blocks, incorrectly typed variables, etc.) One facet of JPP is essentially a "lint for Java". Other tools, most notably

SRC's ESC [3] and ParaSoft's jtest![8] include similar (even vastly superior, in the case of ESC) functionality. JPP was not designed strictly as a static source checker, and thus is not as comprehensive as these highly focused tools.

JavaNCCS. JavaNCCS is a tool that is provided with the Jacob Java development package[9]. JavaNCSS is a simple command line utility which measures code using two standard source code metrics for the Java programming language. The metrics are collected globally, for each class and/or for each function. The two metrics that JavaNCCS reports are Non-Commenting Source Statements (NCSS) and Cyclomatic Complexity Number (McCabe metric). JPP supports both of these metrics as well as many more. Additionally, JavaNCCS provides its report either from the command line or in a Java window. JPP supports these two output modes as well HTML, XML and LaTeX.

JML. JML is a behavioral interface specification language for Java developed at Iowa State University [10]. JPP only provides support for DBC-related specification constructs. JPP has very different goals (solely interface specification for test harness generation) than the JML work (full behavioral specification for validation). Thus, while the two are related because they both deal with specification languages, JML is the much more complete (and complex) of the two.

5.2 Future Work

Version 1.0 of JPP provides a subset of the functionality described in this document. More specifically, JPP version 1.0 supports:

- Parsing and validation of all Javadoc tags included in [7].
- Transformation of legitimate Java code DBC specifications into runtime test code.
- Java pretty printing to the syntax specification in [7].

Some fine-tuning of existing functionality is necessary, most of which are due to idiosyncrasies of the Java grammars and parser generators. This tuning includes:

- Support for documentation comments for inner classes.
- Support for multi-line documentation comments.
- User-customized tag extensions (since the parser depends upon such entities).

The remaining functionality we plan to add is fully described in this document.

[8] http://www.parasoft.com/products/jtest/index.htm
[9] http://mats.gmd.de:8080/clemens/jacob/

References

[1] A.B. Binkley and S.R. Schach. A comparison of sixteen quality metrics for object-oriented design. *Information Processing Letters*, 58(6):271–275, 1996.

[2] S.R. Chidamber and C.F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[3] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice*, pages 1–9. ACM (SIGSOFT), January 1996. The first brief on ESC from SRC.

[4] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *European Conference on Object-Oriented Programming/ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 25/10 of *ACM SIGPLAN Notices*, pages 169–180. ACM SIGPLAN: Programming Languages, ACM Press and Addison-Wesley Publishing Company, October 1990.

[5] B. Henderson-Sellers, L.L. Constantine, and I.M. Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3(3):143–158, 1996.

[6] IBM et al. *Object Constraint Language Specification, version 1.1.* The UML 1.1 Consortium, September 1997.

[7] Joseph R. Kiniry. *The Infospheres Java Coding Standard.* The Infospheres Group, Deparment of Computer Science, California Institute of Technology, 1997.

[8] Joseph R. Kiniry. IDebug: An advanced debugging framework for Java. California Institute of Technology Technical Report CS-TR-98-16, California Institute of Technology, November 1998.

[9] Reto Kramer. iContract – the java design by contract tool. In *Proceedings, TOOLS '98*, volume 26 of *TOOLS Conference Series.* IEEE Computer Society, 1998.

[10] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06a, Iowa State University, Department of Computer Science, July 1998.

[11] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996.

[12] Bertrand Meyer. *Advances in Object-Oriented Software Engineering*, chapter Design by Contract. Prentice-Hall, Inc., 1992.

[13] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.

[14] Terence Parr. *ANTLR Reference Manual*. MageLang Institute, document version 2.4.0 edition, September 1998.

[15] Sriram Sankar. An efficient top-down parsing algorithm for general context-free grammars. Technical Report CSL-TR-93-562, Stanford University, February 1993. Early work that lead to JavaCC and MetaMata work.

[16] The JavaCC Team. *The Java Compiler Compiler (JavaCC), version 0.6.1.* SunTest and MetaMata, February 1997.

[17] R. Whitty. Object-oriented metrics: An annotated bibliography. *ACM SIGPLAN Notices*, 31(4):45–75, 1996.

A Example Documentation Comment Block.

The following code block shows the various Javadoc tags that can be used in a documentation comment when using the Infospheres Java Coding Standard.

```
/**
 * This is a large Javadoc comment block that contains every Javadoc
 * tag currently in use by us.  It is used for a tag reference and a
 * cut-and-paste source.
 *
 * @version Version-string CVS-Date-tag
 * @author Author Name
 * @history Description.
 *
 * @bug Description of the bug.
 * @review Username Description.
 * @todo Username Description.
 *
 * @concurrency (SEQUENTIAL | GUARDED | CONCURRENT) Semantics description.
 * @precondition (Expression) Description.
 * @requires (Expression) Description.
 *
 * @ensures (Expression) Description.
 * @generates (Expression) [Optional Description]
 * @modifies (SINGLE-ASSIGNMENT | QUERY | Expression) Description.
 * @postcondition (Expression) Description.
 * @invariant (Expression) Description.
 *
 * @exception FullyQualifiedExceptionName IF (Expression) Description.
 * @param ParameterName Description.
 * @param ParameterName WHERE (Expression) Description.
 * @return Description.
 *
 * @deprecated Reference to replacement API.
 * @since Version-tag.
 *
 * @hides FullObject.AttributeName [Optional Description]
 * @overrides FullPackageObject.MethodName [Optional Description]
 *
 * @equivalent (Expression | Code reference)
 * @example Description.
 * @see (Description | URL | FullyQualifiedClassname | Classname |
 * Classname#methodName(parameters))
 *
 * @design Description.
 *
 * @references (Expression) [Optional Description]
 * @uses (Expression) [Optional Description]
 *
 * @guard (Expression) [Optional Description]
* @values (Expression) [Optional Description]
**/
```