Fix-it: An Extensible Code Auto-Fix Component in Review Bot

Vipin Balachandran
VMware
Bangalore, India
vbala@vmware.com

***Abstract*—Coding standard violations, defect patterns and non-conformance to best practices are abundant in checked-in source code. This often leads to unmaintainable code and potential bugs in later stages of software life cycle. It is important to detect and correct these issues early in the development cycle, when it is less expensive to fix. Even though static analysis techniques such as tool-assisted code review are effective in addressing this problem, there is significant amount of human effort involved in identifying the source code issues and fixing it. Review Bot is a tool designed to reduce the human effort and improve the quality in code reviews by generating automatic reviews using static analysis output. In this paper, we propose an extension to Review Bot– addition of a component called Fix-it for the auto-correction of various source code issues using Abstract Syntax Tree (AST) transformations. Fix-it uses built-in fixes to automatically fix various issues reported by the auto-reviewer component in Review Bot, thereby reducing the human effort to greater extent. Fix-it is designed to be highly extensible– users can add support for the detection of new defect patterns using `XPath` and `XQuery` and provide fixes for it based on AST transformations written in a high-level programming language. It allows the user to treat the AST as a DOM tree and run `XQuery` UPDATE expressions to perform AST transformations as part of a fix. Fix-it also includes a designer application which enables Review Bot administrators to design new defect patterns and fixes. The developer feedback on a stand-alone prototype indicates the possibility of significant human effort reduction in code reviews using Fix-it.**

I. Introduction

Coding standard violations and defect patterns are abundant in checked-in code [1]. The presence of these source code issues lead to unmaintainable code and bugs in later stages of software life cycle, when it is more expensive to fix. Even though static analysis techniques such as peer code review and automatic static analysis are considered to be effective for solving this problem, as observed in [1], they are not highly effective in practice due to: *a*) significant human effort, *b*) difficulty in validating code against a lengthy coding standard and *c*) prioritization of logic verification in favor of enforcing style and best practices.

In [1], the author proposed a tool called Review Bot, which is an extension to the open-source code review tool Review Board [2]. Review Bot uses output of multiple static analysis tools to automatically post a code review, which covers majority of coding standard violations and common defect patterns. It also recommends appropriate code reviewers based on change history of source code lines. Even though the automatic reviews reduce the human effort and improve the overall code review quality, the developer still has to analyze the issues reported in automatic review, find out potential fixes and re-submit the code for review. Considering the fact that fixes for a subset of coding standard violations and defect patterns can be automated, we propose a pre-processing step before creating an automatic review from static analysis output. This pre-processing step aims to fix automatically a large subset of source code issues. Post this step, there will be less number of source code issues for the developer to fix, which would improve the productivity. As indicated by the user study in [1], majority of source code issues in automatic reviews are due to lack of understanding of coding standard rules and best practices, and the developers were willing to address majority of these issues. This implies the possibility of significant human effort reduction by the integration of automatic source code correction with code review.

Towards this goal of automatic correction of source code issues, we propose a new component called Fix-it in Review Bot. Fix-it internally maintains the source code as Abstract Syntax Tree (AST) and uses AST transformations for auto-correction. It uses built-in fixes performing AST transformations to automatically fix the various issues reported by the auto-reviewer component in Review Bot. Fix-it allows the users to treat the AST as an XML DOM tree and supports extensibility in the form of custom defect patterns (AST structure pattern) written in `XPath` or `XQuery` [3]. It also supports extensibility in the form of custom fixes (performing AST transformations) written in `Java` or `XQuery` UPDATE expressions. Even though the current implementation supports only `Java`, the architecture is generic to support code written in any programming language which has an AST parser. Fix-it also includes a designer application to facilitate Review Bot administrators to create new defect patterns and fixes.

Refactoring is another task where significant human effort is involved. Even though there are refactoring tools such as TXL [4], Stratego/XT [5] etc., which support extensibility in terms of rules written in a tool-specific language, they are often under-utilized. Emerson et al. [6] reported that 90% of the refactorings are manual without using any refactoring tool. One of the reasons for this under-utilization is the learning curve involved in understanding these tools [7]. Fix-it supports custom fixes which can apply a particular automatic refactoring. Since Fix-it is used in a centeralized way, developers need

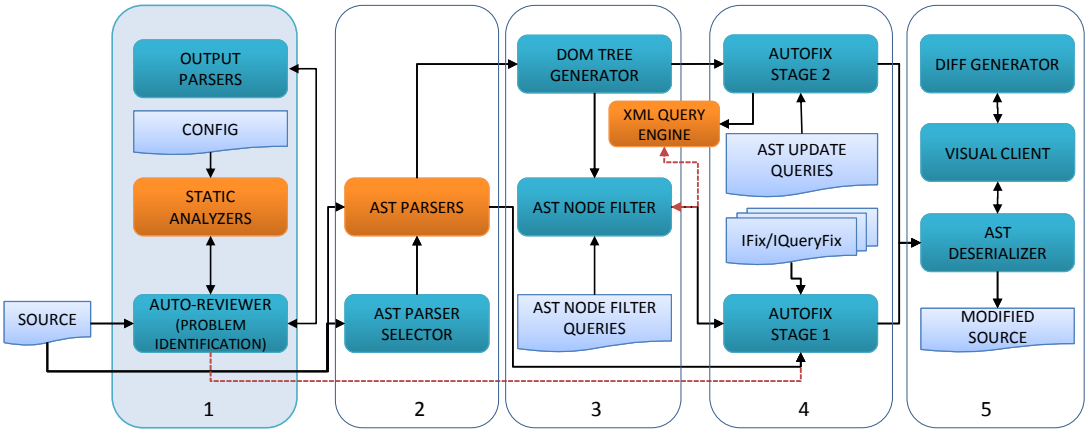


Fig. 1. Fix-it architecture

not learn how to write the refactoring code.

It could be argued that the developers themselves can run static analysis tools within their development environment and manually fix the issues before submitting the changelist for code review. As mentioned in [1], it might be difficult to enforce such a practice. The same argument applies for the use of refactoring tools before code review submission. Also, it would be easier to apply a common configuration for these tools when used in a centralized manner.

The design of Fix-it is motivated by the following observations.

- ASTs are convenient representation of a source code to apply transformations.
- ASTs can be modeled as a DOM tree.
- `XML` and its query languages are widely adopted.

The rest of the paper is organized as follows. In Section II, we discuss Fix-it architecture followed by its extensibility in Section III. Section IV discusses the Fix-it designer application. Section V contains developer feedback on fixes possible for a subset of static analysis rules enabled in Review Bot. The related work is in Section VI and we conclude in Section VII.

II. Fix-it Details

A. Fix-it Architecture

The architecture of Fix-it is shown in Figure 1. The modules within Fix-it and the modules in Review Bot which it interacts with are grouped into the following functional stages:

1) **Problem Identification**
   The modules in this stage are already part of Review Bot. These modules are responsible for identifying the issues in the source code using automatic static analysis tools such as FindBugs [8]. The *Auto-Reviewer* invokes various *Static Analyzers*, which in-turn analyzes the code for coding standard violations and defect patterns. The *Config* module supplies the necessary configuration (rules/checks to run, customized error/warning messages etc.) for the static analyzers. The output format varies from one static analyzer to another; hence the auto reviewer uses the *Output Parsers* module to convert the various static analyzer output to a common format.
   It should be noted that a stand-alone version of Fix-it can be made by implementing these modules separately. In such a case, the auto-reviewer could be relabeled as problem identification module.
2) **AST Generation**
   In this stage, the *AST Parser Selector* selects an AST parser based on the source language and an AST is constructed. The current implementation uses `ASTParser` in Eclipse JDT core [9]. In an alternate implementation, one could also use a collection of ANTLR parsers [10] to add support for multiple programming languages.
3) **DOM Tree Generation and Node Filtering**
   The *DOM Tree Generator* creates a DOM tree adapter which wraps the input AST in such a way that the updates on the DOM tree are translated to updates on the underlying AST. The *AST Node Filter* uses the *XML Query Engine* to evaluate `XPath` and `XQuery` expressions (*AST Node Filter Queries*) to filter out AST nodes which match user-defined defect patterns. The current implementation uses Saxon [11] as the query evaluation engine.
4) **Auto-correction** This stage uses a collection of automatic fixes to fix some of the source code issues identified in Stage 1 and defect patterns identified in Stage 3. The *Autofix Stage*$_1$ module selects and applies a fix (written in `Java`) based on the source code issue type or AST node filter query. The *Autofix Stage*$_2$ module applies the fixes written as `XQuery` UPDATE expressions (*AST Update Queries*). It transforms the AST by evaluating the queries on the DOM tree wrapping the AST.
5) **User Interaction** This is the final stage– it generates the modified source code from the transformed AST (*AST Deserializer*) and presents the identified problems and the modified source code to the user (*Visual Client*). The visual client uses a *Diff Generator* to generate the side-by-side line diff and provides options to inspect the problems, corresponding fixes and to selectively apply a subset of fixes to the original source file.

FIX-IT($srcText$)

```
1  P = FINDPROBLEMS(srcText) // Stage1
2  // Stage2
3  parser = RESOLVE-AST-PARSER(srcText)
4  ast = parser.PARSE(srcText)
5  // Stage3
6  domTree = GENERATE-DOM-TREE(ast)
7  i = 1
8  for each q ∈ AST-NODE-FILTER-QUERIES()
9       result = EVALUATE-QUERY(domTree, q)
10      C[i] = CREATE-FIX-CONTEXT(result, q, ast)
11      i = i + 1
12 // Stage4
13 for j = 1 to P.length
14      C[i] = CREATE-FIX-CONTEXT(P[j], ast)
15      i = i + 1
16 for i = 1 to C.length
17      fix = RESOLVE-FIX(C[i])
18      fix.APPLY(C[i])
19 for each q ∈ AST-UPDATE-QUERIES()
20      EVALUATE-QUERY(domTree, q)
21 // Stage5
22 modifiedSrc = DESERIALIZE(ast)
23 return VISUAL-CLIENT(srcText, modifiedSrc)
```

Fig. 2. Algorithm - Fix-it

B. Algorithm

The fix-it algorithm is in Figure 2. The problems in the source code are found in line 1. The AST parser is resolved based on the source language and the AST is generated in lines 3-4. The **for** loop in line 8 filters AST nodes by evaluating `XPath` and `XQuery` expressions (line 9) on the AST DOM tree adapter (created in line 6) and creates fix context for each of the query results (line 10). The fix context for each of the problems identified in Stage$_1$ is created in line 13. The **for** loop in line 16 determines the fix (line 17) for each of the fix contexts and applies the fix (line 18). In line 19, the AST is transformed by evaluating `XQuery` UPDATE expressions. The transformed AST is deserialized in line 22 and the visual client is created and returned in line 23.

III. EXTENDING FIX-IT

A. Support for New Static Analyzers

New static analyzers can be added in Stage$_1$ if it is possible to write an output parser for it. Most of the static analyzers produce output with some structure and an output parser is feasible in most cases. The current implementation of Review Bot supports only `Java` and uses FindBugs, Checkstyle [12] and PMD [13] for static analysis. It is a future work to add support for other languages; the main challenge in Fix-it will be to modify the fix stages to work with different types of ASTs.

```
context.getUnit().accept(new ASTVisitor() {
   @Override
   public boolean visit(MethodDeclaration node) {
      for (Object obj : node.modifiers()) {
         Modifier m = (Modifier) obj;
         if (m.getStartPosition()==context.getPosition()) {
            context.getASTRewrite().remove(m, null);
            break;
         }
      }
      return false;
   }
});
```

Listing 1. Fix for Checkstyle's `RedundantModifier` check

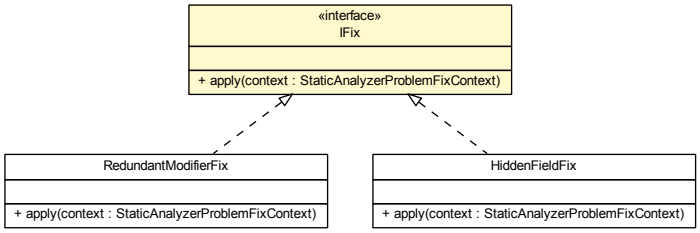


Fig. 3. Part of `IFix` type hierarchy

B. Adding a New fix

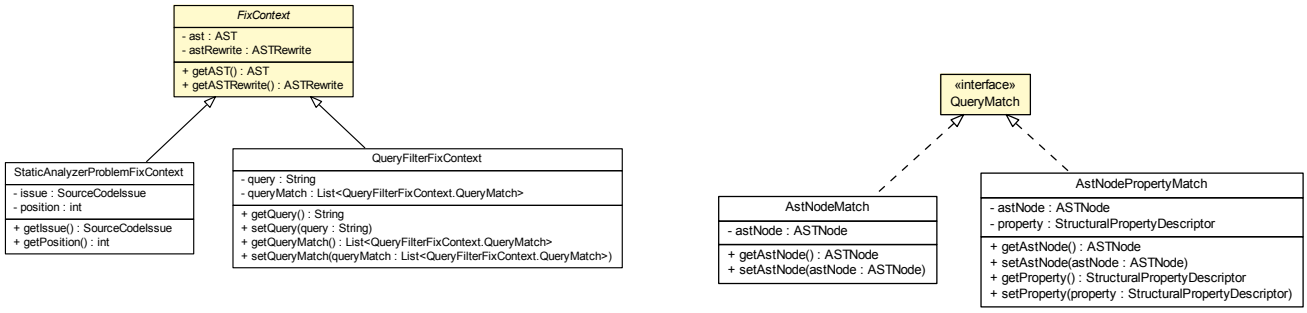
If there is no existing fix for a problem identified in Stage$_1$, users can write a fix in `Java` and add to the collection of fixes. This is done by implementing the `IFix` interface (Figure 3) and associating the `.class` file with the problem ID. The details of the identified problem will be passed in as an instance of `StaticAnalyzerProblemFixContext` (Figure 4(a)). As an example, Listing 1 is a snippet from the built-in `RedundantModifierFix` for the `RedundantModifier` check (to detect redundant modifiers; for example, `public`, `static`, `final` modifiers are redundant for a variable declaration in an `interface`) in Checkstyle. Users can follow the same pattern of visiting interested node type (`MethodInvocation` in `RedundantModifierFix`), checking if the node covers the problem identified and finally applying changes to the node. The Fix-it designer (Section IV) can be used to identify the AST node type associated with a specific problem.

C. Support for Custom Defects

Users can write `XPath` or `XQuery` expressions to detect custom defect patterns. For example, the following `XPath`

```
@Override
public void apply(QueryFilterFixContext context) {
   for (QueryMatch m : context.getQueryMatch()) {
      assert m instanceof AstNodeMatch;
      ASTNode node = ((AstNodeMatch) m).getAstNode();
      assert node instanceof NumberLiteral;
      NumberLiteral literal = (NumberLiteral) node;
      context.getASTRewrite().set(literal, NumberLiteral.
          TOKEN_PROPERTY,
                "0" + literal.getToken(), null);
   }
}
```

Listing 2. Fix for the violation of missing digit before decimal point



(a) `FixContext` type hierarchy

(b) `QueryMatch` type hierarchy

Fig. 4. Fix context types

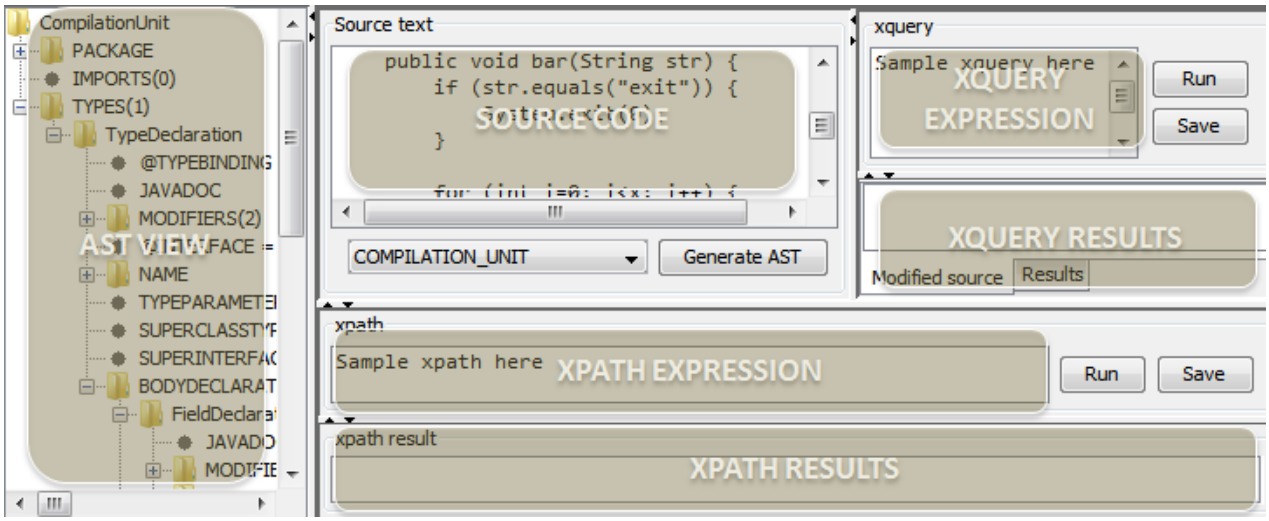


Fig. 5. Fix-it designer

expression when evaluated on a DOM tree adapter of an AST detects violations of the coding standard rule: "Floating point constants should always be written with a digit before the decimal point.".

```
//VariableDeclarationFragment[../..//@PRIMITIVETYPECODE="
    float" and fn:starts-with(.//NumberLiteral/@TOKEN, ".")
    ]//NumberLiteral
```

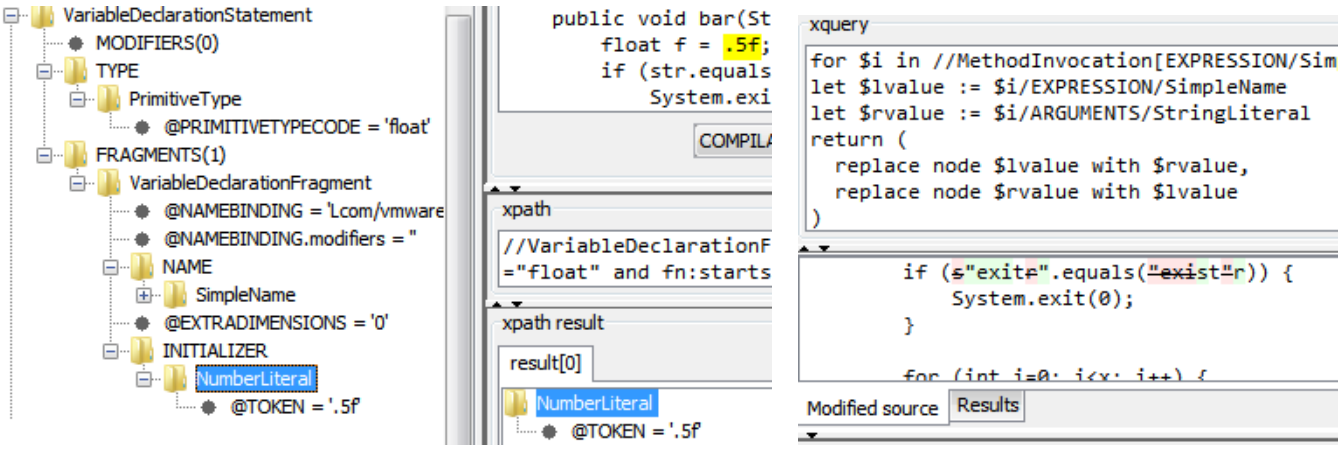
Filter queries like the above can be designed using the Fix-it designer application (Section IV-A). The results of the filter query are used to construct a fix context (instance of `QueryFilterContext`, see Figures 4(a) & 4(b)), which is then passed to the AutoFix Stage$_1$ module, where a fix (implementation of `IQueryFilterFix` which has a method `accept` with an argument of type `QueryFilterContext`) is resolved based on the query string and applied. The fix in the case of the above filter query involves modifying the `TOKEN` property of the `NumberLiteral` AST node (Listing 2). If the defect pattern is complicated to be expressed in `XPath` or `XQuery`, users can input the query as '`/`' which matches the root of the AST and the corresponding `IQueryFilterFix` can traverse the AST to find the desired defect pattern and fix it.

```
let $pattern := "^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$"
for $i in //FieldDeclaration
            [MODIFIERS/Modifier[@KEYWORD="final"]]
               //SimpleName[not(fn:matches(@IDENTIFIER, $
                  pattern))]
let $b := $i/@NAMEBINDING
let $id := fn:upper-case($i/@IDENTIFIER)
return (
  replace value of node $i/@IDENTIFIER with $id,
  for $j in //STATEMENTS//SimpleName[@NAMEBINDING=$b]
       return replace value of node $j/@IDENTIFIER with $id
)
```

Listing 3. Fix for Checkstyle's ConstantName check

D. Writing a Fix in `XQuery`

In cases where a fix is possible by simply deleting a node in the AST or by replacing the value of an AST node property, users can provide an `XQuery` UPDATE expression which modifies the AST DOM tree adapter. For example, the expression in Listing 3 provides a fix for the `ConstantName` check in Checkstyle which mandates that a constant identifier contains only upper-case letters, digits and underscores.



(a) AST view and filter based on `XPath` expression

(b) AST transformation using `XQuery`

Fig. 6. Designing fixes using `XPath` and `XQuery`

IV. Fix-it Designer

The designer application provides an interface for designing AST node filter queries and AST DOM tree transformation queries. Figure 5 shows the main window. The *AST view* pane displays the AST of the code snippet entered in *Source Code* pane. For example, the AST sub-tree corresponding to the variable declaration `float f = .5f` is shown in Figure 6(a). The highlighted AST Node corresponds to the code fragment "`f = .5f`". Each of the AST nodes can have zero or more attributes (names prefixed with @) and zero or more child elements (nodes with name in upper-case letters). Each of the child elements can have zero or more AST nodes as its children; the child count is in parentheses after the node name. The `*BINDING` attributes are special attributes which doesn't have a structural significance and is used for cross-referencing between AST nodes. For example, any `SimpleName` node corresponding to the `float` variable `f` (say the node representing `f` in the `ExpressionStatement` `f++`) will have the same `NAMEBINDING` value as that of its `VariableDeclarationFragment` (node representing its declaration). Listing 3 uses this idea to find the references of the constant identifier.

A. Designing AST Node Filter

A node filter can be written in either `XPath` or `XQuery` considering the AST as an XML DOM tree. Figure 6(a) shows the results of evaluating the following `XPath` expression on the AST shown in the AST view pane.

```
//VariableDeclarationFragment[../..//@PRIMITIVETYPECODE="
    float" and fn:starts-with(.//NumberLiteral/@TOKEN, ".")
    ]//NumberLiteral
```

In this case, the user is interested in all floating point literals starting with a decimal point. The expression returns all variable declaration nodes with float type if the corresponding literal starts with a decimal point. Listing 2 is a fix corresponding to this filter.

If the filter query is complicated, `XQuery` FLWOR expression can be used instead of `XPath`. The following example shows the filter for unused method parameter. The query returns all the variable declaration nodes which are method parameters if they are not referenced at least once in the method body.

```
for $i in //MethodDeclaration/PARAMETERS/
    SingleVariableDeclaration
let $b := $i/@NAMEBINDING
where (
   fn:empty($i/../../BODY/Block//SimpleName
      [@NAMEBINDING = $b])
)
return $i
```

B. Designing AST update queries

Figure 6(b) shows the output of evaluating the following `XQuery` UPDATE expression.

```
for $i in //MethodInvocation[EXPRESSION/SimpleName and
    ARGUMENTS/StringLiteral and @METHODBINDING = "Ljava/
    lang/String;.equals(Ljava/lang/Object;)Z"]
let $lvalue := $i/EXPRESSION/SimpleName
let $rvalue := $i/ARGUMENTS/StringLiteral
return (
  replace node $lvalue with $rvalue,
  replace node $rvalue with $lvalue
)
```

This UPDATE expression fixes the violation of Checkstyle's EqualsAvoidNull check which ensures that the string literal is on the left side of `equals()` comparison with a `String` instance variable. On evaluating this expression, it finds all `String.equals` method invocation nodes with variable name on left side and string literal on right side and swap them.

The next example provides the fix for the case where a floating point literal starts with a decimal point.

```
for $i in //VariableDeclarationFragment[../..//
    @PRIMITIVETYPECODE="float" and fn:starts-with(.//
    NumberLiteral/@TOKEN, ".")]//NumberLiteral
let $v := fn:concat("0", $i/@TOKEN)
return replace value of node $i/@TOKEN with $v
```

V. Results

A. Feedback on Checkstyle Fixes

As per the results in [1], on average, 86% of comments in a Review Bot automatic review was generated by Checkstyle.

TABLE I
DEVELOPER FEEDBACK ON CHECKSTYLE FIXES

Checkstyle Module	#Rules enabled	#Rules with fixes	#Rules with fixes (advanced features)
Annotations	2	2	2
Block Checks	5	2	5
Class Design	6	4	5
Coding	30	16	19
Duplicate Code	1	0	0
Headers	1	1	1
Imports	6	4	4
Javadoc Comments	4	0	0
Metrics	5	1	1
Miscellaneous	6	3	6
Modifiers	2	2	2
Naming Conventions	11	5	10
Regexp	4	3	3
Size Violations	6	0	1
Whitespace	12	0	12
Sum	101	43	71

In an attempt to estimate the effort reduction due to Fix-it, we requested an experienced developer who is proficient in Java and XML technologies to check whether fixes can be provided for various Checkstyle rules enabled in Review Bot. The developer was provided with all the details of Fix-it and ways to extend it and feedback was requested on the following:

- For each of the rules enabled, is it possible to provide a fix with the current features in Fix-it?
- For rules where a fix is not possible, is it possible to provide a fix if Fix-it supports complex refactorings involving multiple files and manipulating the source code text directly?

The feedback grouped by functionality is given in Table I. The ability to fix 43% of issues with the current features and 70% with advanced features (planned for future) implies significant effort reduction during code review. Details about the grouping can be found in Checkstyle online documentation.

VI. RELATED WORK

The Eclipse IDE [14] has support for automatic code formatting and cleanup. It also supports automatic code correction in the form of quickfixes. Even though the formatting and cleanup features can be user configured, it is not possible to extend any of these features. Static analysis tools like FindBugs and Checkstyle support extensibility by writing new detectors in Java. The static analysis tool PMD supports extensibility via Java or by writing AST defect patterns using `XPath`, which is quite similar to the node filtering use case in Fix-it designer. However, there is no support for automatic correction in any of these tools. AppPerfect [15] is a static analyzer which uses around 750 rules to detect various source code issues and automatically fixes 180 issues, but lacks extensibility. Also, as mentioned in [16], static analysis tools will receive limited use depending on how they are integrated with the development process. There are several refactoring tools which support user extensibility [4], [5], [17], [18]; however these type of tools are often under utilized [6] and it might be difficult to enforce its usage as in the case of static analysis tools. The utilization of refactoring tools could be improved by a tight integration with the development process, like the way we proposed in this paper.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a code auto-correction component called Fix-it in Review Bot in order to reduce the human effort in code review to a significant extent and to improve the code quality. We discussed various ways to extend Fix-it and the designer tool to support extensibility. Support for writing complex refactorings involving multiple files, hooking into external frameworks during autofix stages and adding new languages etc. are some of the future works planned.

VIII. DOWNLOADS

A stand-alone prototype version of Fix-it and the demonstration video can be downloaded at http://bit.ly/12fIiu6.

REFERENCES

[1] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 931–940.

[2] "Review Board," http://www.reviewboard.org/.

[3] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler, "Xquery 1.0 and xpath 2.0 formal semantics," *W3C recommendation*, vol. 23, 2007.

[4] J. R. Cordy, "Source transformation, analysis and generation in txl," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, ser. PEPM '06. New York, NY, USA: ACM, 2006, pp. 1–11.

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/xt 0.17. a language and toolset for program transformation," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 52–70, Jun. 2008.

[6] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 5 –18, jan.-feb. 2012.

[7] D. Campbell and M. Miller, "Designing refactoring tools for developers," in *Proceedings of the 2nd Workshop on Refactoring Tools*, ser. WRT '08. New York, NY, USA: ACM, 2008, pp. 9:1–9:2.

[8] "FindBugs," http://findbugs.sourceforge.net.

[9] "JDT Core Component," http://www.eclipse.org/jdt/core/index.php.

[10] "ANTLR," http://www.antlr.org/.

[11] "SAXON - The XSLT and XQuery Processor," http://saxon.sourceforge.net/.

[12] "Checkstyle," http://checkstyle.sourceforge.net/.

[13] "PMD," http://pmd.sourceforge.net/.

[14] "Eclipse," http://www.eclipse.org/.

[15] "AppPerfect," http://www.appperfect.com/.

[16] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 241–252.

[17] H. Li and S. Thompson, "Let's make refactoring tools user-extensible!" in *Proceedings of the Fifth Workshop on Refactoring Tools*, ser. WRT '12. New York, NY, USA: ACM, 2012, pp. 32–39.

[18] K. Maruyama and S. Yamamoto, "Design and implementation of an extensible and modifiable refactoring tool," in *Proceedings of the 13th International Workshop on Program Comprehension*, ser. IWPC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 195–204.