Abstract
Refactoring program code preserves its semantics, even while changing its structure. Program comments do not carry semantics, so by definition an automated refactoring tool can remove them to be formally correct. However, if carefully crafted formatting or comments disappear while refactoring code, developers will be unhappy. Therefore, a useful refactoring tool should keep comments in refactored code and move them with code they belong to.

We present different approaches on how to keep source code comments when refactoring code. Usually comments are no longer present in an abstract syntax tree (AST) used by a refactoring tool and our heuristics to associate comments with nodes in an AST. The different attempts on gathering comments and implementing that association are shown and evaluated.

Details of comment handling in different refactoring plug-ins, we have implemented for Eclipse, are given as well as brief comparison with comment handling when refactoring as implemented by the Eclipse java development tools (JDT).

We hope, that this paper enables others as well to implement comment-preserving refactoring tools for more languages and IDEs. Every programmer, language and IDE deserves good refactoring support or they might become endangered.

1. Introduction
Refactoring of program code preserves its semantics, even while changing its structure. Program comments do not carry semantics, so by definition an automated refactoring tool does not need to deal with comments to be formally correct. However, if carefully crafted formatting or comments disappear while refactoring code developers will be unhappy. Therefore, a useful refactoring tool should keep comments in refactored code and move them with code they belong to.

We present different approaches on how to keep source code comments that are usually no longer present in an abstract syntax tree (AST) used by a refactoring tool and our heuristics to associate comments with nodes in an AST they belong to as well as keep the code’s original formatting as well as possible.

1.1 Relevance
Most programming languages define blanks, tabs and line feeds as white space, that are irrelevant for a program’s semantics. Often comments are also considered as such semantic-free white space. Nevertheless readability and understandability heavily depends on source code’s formatting and a program’s comments.

Today, developers use integrated development environments (IDE) to work with source code. Many IDEs provide features for automatically creating and manipulating code. Refactoring and generation of boilerplate source code within an IDE and other coding assistance features must at first priority create syntactically correct code and secondly create code that has semantics intended by the developer.

Thus, coding assistance is quite syntax-oriented as you don’t want an assisting functionality of your IDE (integrated development environment) to produce code not accepted by your compiler.

Such development aids today are often implemented as plug-ins to general IDE platforms, such as Eclipse. We implemented refactoring support plug-ins for Eclipse CDT, RDT and other Eclipse-based IDEs.

IDE plug-ins that support programmers must create correct code. However, developers expect more from such tools: A plug-in touching program code, should not alter the source’s general appearance. Thus the modified source
code mustn’t be reformatted. I.e., no unexpected line breaks are wanted. Therefore, source code should look like it did before applying a change by a plug-in. This impact can be minimized if configurable source code formatting is part of the IDEs functionality and a refactoring plug-in can re-use that functionality for generated code.

1.2 Extract Method Example

Figure 1 shows the extraction of some ruby code into its own method. The dark region in the first function represents the selected text. The desired outcome of this extraction are the next two functions. The last two functions show an unsuccessful attempt: the code is still valid, but we lost the comment and the formatting looks horrible.

Unfortunately there’s usually more than just spaces, tabs and line breaks - the comments. Their handling cannot be covered or defined by general guidelines, especially because they have a content which is important. While being irrelevant for the compiler, as comments are usually considered as whitespace, the programmer usually wrote and placed them for reason and wants them to be untouched relative to the according code.

Therefore a careful treatment of comments should be one aspect of a reliable code modification tool.

2. Comments, Formatting and the AST

A compiler or interpreter usually creates an abstract syntax tree (AST) to keep information about a program’s code structure. Often the AST contains original line numbers to map error messages and debugging information to the source code.

Modern IDE implementations depend on parsing information stored in an AST to provide program structure information. However, association of program elements (nodes in the AST) to source code lines is to coarse grained to allow highlighting of structural elements or the extraction of statements or expressions for refactoring. Even a fancy editor might need exact information about the position of names and keywords in the source code text to allow syntax highlighting and features like auto completion.

IDE plug-ins that automatically create and modify program code, like refactoring tools, on the one hand depend on the AST for structure information to keep code syntactically well structured. For example, the definition of an identifier and its usage is represented in the AST or accompanying data structures. This allows to recognize all places that need to be adjusted in case of a rename refactoring. On the other hand, these tools require exact source positions in the format of lines and columns or character offsets and the lengths of AST elements to exactly determine positions for inserting or removing code snippets, especially when more complex refactorings than just rename of an identifier are performed by transforming the AST and re-generating source code from it.

A compiler’s or interpreter’s AST often lacks this exact source code relationship, because it is not required. An IDE editor with syntax highlighting and auto completion can work without absolute accuracy. However, a refactoring tool manipulating code, often in several places simultaneously must rely on exact location information of a syntactical element. Otherwise, the result of a code transformation might be syntactically incorrect or just a mess. Another use of the AST is to (re-)generate code based on its AST representation. This should be done as close to the original “concrete” syntax that tends to get lost in parsing.

A typical Eclipse-based IDE uses a parser based on a language’s compiler or interpreter to provide the AST and structure information needed for its tools. Unfortunately many such implementations that do not yet provide refactoring support, lack the accuracy of source element position information or AST “concreteness” required for refactoring automation. Adjusting the language’s parser to provide more accurate source code positions is often an initial task to enable refactoring automation.

While exact source element positions in an AST provide most of a code’s formatting, a program’s comments usually are not represented in the AST, since comments are of no importance for compilers and interpreters.
Thus, the refactoring plug-in developer is on his own. While creating many extensions for IDEs of several languages, we’ve collected some experience on how to retain comments and re-generate code while refactoring, which we want to share. This includes techniques on detecting and keeping the comments available for the refactoring plug-in, associating comments with elements in the AST and keeping them around in expected places while refactoring code.

2.1 Gathering Comments

First of all, since comments are not part of the AST, a prerequisite for a refactoring plug-in is to detect and collect the comments with their positions.

There are several ways to detect comments in the code. For example, the language specific comment pattern, i.e. as a regular expression, that is matched against all source files separately from the parsing process. When using this recognition method you mustn’t forget that those patterns can occur at places where they should not be treated as comments, for example embedded in strings. A better way would be to extend the IDE’s scanner or parser to collect all comments while parsing the source code. This reuses the comment detection features already available and eliminates the risk for false positives or missed comments.

In our experience the best way for comment detection is to use the IDE’s scanner and parser which is already in the IDE. This saves time, since the parsing is done anyway and files don’t have to be scanned twice.

However, while ideally the existing IDE parser is already collecting comments with their source position for later use by other IDE plug-ins, this is not always the case. So in addition to content and line numbers of comments, their exact source position as well as their association with AST’s elements they are describing is important.

For the former, we often needed or change the scanner and/or parser of a language to keep comments around. This meant, providing a refactoring plug-in, the core support for the corresponding programming language in the IDE needed to be augmented. Sometimes even the implementation of the underlying languages used by the IDE needed to change, as was the case with JRuby and Groovy. To avoid interpreter or parser performance impact the collection of comments was kept separately from the AST and could be turned on and off as needed.

The heuristics applied for the latter problem of associating comments with syntactical elements in the AST are the main contribution of this paper.

3. Associating Comments with AST Elements

After being able to detect and collect comments while parsing source code, the next step is to associate these comments with syntax elements such as declarations, definitions and statements within the AST. These syntax elements are represented as AST nodes. The association of comments with AST nodes enables the refactoring tool to re-generate the commented code from the AST even when the corresponding syntax elements are moved within the AST according to a refactoring’s AST transformation.

A major problem in handling comments correctly is the determination where a comment belongs to and which node of the AST it should be associated with. Several different approaches are imaginable.

3.1 Line and Block Comments

First, we want to distinguish between languages that possess a line (end) commenting style, such as Ruby, and languages that allow block comments and line comments, such as Java and C++. In Ruby a ’#’ symbol starts a comment that ends at the end of the line. Multi-line comments are represented by several commented lines. In Java and C++ comments can be either given as a block comment with a start and end indication /* comment */ or by using ‘/’ for a line comment. There cannot be code on the same line after a line comment.

This distinction between line and block comments is not relevant for associating comments with nodes, but must be considered while (re-)generating code from an AST with associated comments. There a line comment might be generated within a syntactical element, that would require more "real" code on the same line that already has a line comment. A naive implementation of a code generator might just break the program. Depending on a language’s syntax some re-ordering of AST nodes with respect to the comment might be needed, if the remaining nodes of a statement cannot be generated just on the following source line. This is especially true, with scripting languages, where a line break usually is also statement break.

Every language has its own symbols for marking comments. Comments can also be used for different functions such as documentation (like JavaDoc), explanations or tasks.

To sum up, all these different sorts of comments can be put into two groups:

- line comments
- whitespace comments

According to our own definition a line comment is dedicated to a line and after the comment symbol there is nothing else than this comment on the same line. It is then terminated by the line delimiter. A whitespace comment is what we call a comment which can be written instead of a whitespace in the middle of any code.

For example

- Ruby - Line comment
  
  ```ruby
  #not possible to write code after a hash
  puts "Hello World" # but it’s possible to write code
  ```

- C++ - Line comment
  
  ```cpp
  ```
/*
 * In C++ the comments that can be in every whitespace are called block comment
 */
int i = /*another block comment*/ 0;

3.2 Definitions

To ease the discussion about comment handling we categorize comments according to their relative position with respect to the associated AST node. In our C++ refactoring plug-in three different categories are used:

- leading comments - Comments before a statement, declaration, or definition.
- trailing comments - Comments right after the AST node on the same line.
- freestanding comments - Comments before a closing brace such as they occur in namespace-, class- and method-definitions or at the end of a file.

Each comment collected will be assigned according to one of these categories and thus be associated with either the following, preceding or surrounding AST node.

The following C++ code shows examples of comments in the different categories:

class Foo
{
  //leading
  public:
    Foo();
    virtual ~Foo(); /*trailing*/
    /* leading */
    void bar();
  private:
    int i;
    //freestanding
};

3.3 Associating Comments with AST nodes

Here we show, how we determine which comment should belong to which AST node and in what position category. Our algorithm is recursively visiting the AST starting with its root node and recursing through children depth-first until all comments are associated with an AST node. For this visiting strategy we can employ the existing AST visitor infrastructure. All unassociated comments are kept in a list sorted by their source position. The visiting strategy guarantees that AST nodes are visited in their source sequence as well and each AST node has its associated source position and range.

For a given AST node, we first check if we are done. If there are still unassociated comments preceding the current node they are associated as leading comments. If the first comment in the list starts on the last line of the current node, it is associated as trailing comment. In the case that the next comment is beyond the current node’s source range, the children of the current node need not be visited, since no comment can be associated with them. In all other cases, that means the first comment is in the range of the current node, the algorithm recurses to the node’s children. If after returning from the children there are comments left that are within the source range of the current node, it is checked if they are trailing the current node on its end line or are freestanding comments within the node’s source range. Both these matching comments are then assigned to the current node and the algorithm returns from the current recursion step. Whenever all comments are assigned the algorithm stops.

If there are unassigned comments remaining while recursion returns to the root node, the remaining comments on the file level are appended as trailing comments to the last node in the AST. Thus no comment gets lost.

3.4 Special Situations

While the above algorithm associates comments with AST nodes to keep them at the appropriate place when the AST is transformed and code is re-generated it can not cover all usage scenarios of comments perfectly.

For example, comments used to deactivate a piece of code will be associated with the following statement or definition as a leading comment. That means, we assume that the comment preceding a definition is an explanation of that definition. This might lead to surprises, for example, when in the following implementation the method Foo::bar() is moved to another class and the commented method Foo::answer() gets moved as well.

```cpp
#include "Foo.h"
#include<iostream>
Foo::Foo()
{
}
Foo::~Foo()
{
}
//Foo::theAnswer()
// std::cout << "42" << std::endl;
//}
Foo::bar()
{
Figure 2. Comment Assignment v2

```cpp
std::cout << "bar" << std::endl;
```

There might be better associations of comments possible, if the semantics of comments’ content could be understood. May be some pattern matching between names of AST nodes and text within an adjacent comment can help associate comments in a context sensitive way according to programmer’s intention.

### 3.5 Representing and Using Comment Association

While the previous section explained how comments can be associated with AST nodes, we give several options on how to actually implement this association in a way that it can be exploited while re-generating code from a transformed AST. We tried several approaches to integrate the comment associations with an AST:

- **invasive**: directly changing the AST classes
- **minimal invasive**: using dynamic proxy objects wrapping existing AST nodes
- **non invasive**: using a hash map to store the association from AST nodes to comments

#### 3.5.1 invasive

The invasive approach by just extending the AST classes was often not an option, since the AST classes usually stem from either the corresponding language’s core implementation (as for example with JRuby or Groovy) or because they are part of a plug-in project we do not have committer rights for and where a large patch changing the many AST classes is not easily accepted as a contribution. In both cases changing performance in other AST applications, such as interpretation or parsing itself are an issue and in the case of an existing plug-in project also the intellectual property (IP) rights can be hurdle taking several months to cross. Another roadblock of the invasive approach that extending the AST node classes not only means adding attributes to hold the associated comments, but also to extend the AST node interface to deal with these associated comments.

#### 3.5.2 minimal invasive

A dynamic language like Ruby would have made the association of comments with AST nodes much easier, but Eclipse plug-ins are implemented in Java and that language doesn’t provide dynamic object or class extension easily.

With this concept of dynamic object extension in the back of our heads and the limitation of being unable to change the existing AST node classes, we tried to apply the Java dynamic proxy objects to solve the class extension problem in a minimal invasive way.

The idea of Java’s dynamic proxies is that the functionality of a node is expanded by wrapping it with a new object through reflection. In this way every time a method on the wrapped node object is called the proxy calls the original method over indirection, can extend its functionality and the calling code remains unchanged. On this way it’s possible to change method behavior and even introduce new methods that are not available in the original node. We used that approach to augment AST nodes with attributes for storing the comments and provided access to these comments.

Based on the dynamic approach we believed we could still reuse the original AST node visitor infrastructure to traverse the augmented AST based on the dynamic proxy objects to re-generate source code including comments from it. The principle looked promising, but it turned out that the implementation task was challenging. Debugging dynamic proxies is almost impossible, because all real method calls happen via reflection. To generate the augmented AST, we used the original visitor architecture with our own visiting strategy. Implementing this approach showed us two major roadblocks, besides the debuggability: First, a huge performance impact because the whole AST had to be cloned and second, the compare method used in some parts of the CDT plug-in didn’t work anymore because it compared object ids which became different because of the proxy objects wrapping the original AST nodes. Thus after a lot of implemen-
tation effort we concluded the dynamic proxy approach as failed.

3.5.3 non invasive

Nevertheless, working with dynamic languages gives more ideas on implementing object augmentation with additional data. We analyzed places where the comments were actually relevant. The one area was the algorithm to associate the comments with AST nodes, the other the re-generation of code based on the AST that should include the comments in the appropriate locations, close to their corresponding AST node. All other areas where AST nodes are used within the IDE’s plug-in need to use the AST nodes untouched.

Therefore, we now store the association in a hash map with AST node references as keys and a collection of comment objects including position information as values. The association algorithm creates entries in that hash map and our code generator employs the hash map to figure out if the currently written AST node requires adding a comment. We only needed to implement two AST visitors, one implementing the association algorithm and the other one creating source code from AST (sub-) trees.

This non-invasive implementation will be part of the CDT release 5.0 as part of the Eclipse Ganymede release in Summer 2008.

3.6 Language Specific Varieties

4. C++

The above general description summarizes most of our implementation of comment retention and association within our refactoring plug-in for the Eclipse C/C++ development tools (CDT). However, collecting comments required adaptation/extension of CDT’s scanner and parser.

Fortunately, we were able to get these small changes to the scanner and the class representing the C++ translations unit accepted as patches to CDT. Otherwise we would not have been able to implement comment retention in our refactoring plug-in completely. The adapted scanner collects all comments with their positions in a list attached to the translation unit object. This object represents the root of the AST and its comment list used by our refactoring plug-in to associate comments with AST nodes as described above.

4.1 Comment Categories Improve Performance

First we believed that our comment categories (leading, trailing, freestanding) could be easily determined again when needed while writing code representing an associated AST node. It turned out that keeping the information if a comment needs to be output before or after the node is far more efficient than calculating that information again when it is needed. Therefore we save the category information together with the comments.

5. Ruby

In a term project and a consecutive diploma thesis at the HSR we’ve created refactoring support for the Eclipse Ruby Development Tools (RDT). During these projects we’ve gathered experience in implementing comment handling from bottom up.

5.1 Environment

RDT embeds JRuby for execution and debugging of Ruby code, so we decided to implement our refactoring support basing on the JRuby AST, which obviously is already available in this environment. As expected JRuby didn’t contain the possibility to present comments in the AST. At this point our challenge started.

5.2 Getting Comments into the AST

Fortunately we didn’t have to start from scratch with JRuby considering the comments. At least the scanner, which creates lexical tokens from the source code, recognized comments but just skipped them for further processing. The parser remains completely ignorant about these skipped source parts.

Our approach evolved with our experience in trying to implement our ideas. Because RDT is not an official Eclipse project and we got in close contact with RDT’s lead developer Chris Williams, we were able to use an invasive approach augmenting the AST to contain comment information. We extended the AST node’s base class to be able to associate comments with each AST node. In this way we have direct access to all comments belonging to an AST node. While re-generating code from the augmented AST, our AST visiting writer will print out the comments just after their associated nodes.

The problem remained that JRuby’s parser was ignorant of comments and JRuby’s scanner was unaware of the AST while detecting a comment.

5.3 Assigning Comments with Grammar Rules

Our first solution focused on defining comment tokens and adapting the parser and its rule set to accept these comment tokens. Because Ruby only defines line comments, the parser accepted a newline token we believed the change would only affect a few basic parser rules.

There was one simple change to the scanner which new had to be able to create a new token which represented a comment. This was quite simple as comment recognition was already implemented. We just had to change the handling of comments from skipping the comment to creating a comment token. Hence our token stream created by the scanner had one more token type to be handled by the parser as we intended.

Subsequently we’ve adapted grammar rules which we expected to be sufficient for adding comments. Soon after creating rules that covered our test cases, new and unexpected
comment positions not acceptable by our modified grammar showed in real code. Adding a set of matching rules satisfied the new requirements in the first place. But more and more new unexpected cases where our grammar needed to accept comments led to a grammar complexity that became unmanageable.

Eventually at the end of the first milestone we decided to abandon the approach of changing the grammar and searched for a better approach to embed comments into the AST.

5.4 Assigning Comments by the Scanner

From the failed approach we’ve learned that it is unforgiving to have an implementation which doesn’t cover unexpected occurrences of comments within the syntax rules. We especially wanted not to break the parsing process when having an unexpected comment node somewhere.

Thus our next approach was scanner-centered. We tried to create comment nodes already in the scanner and assign them to a corresponding token. So in the parser, which creates the effective nodes of the AST, we could check the tokens for assigned comments and pass them to the newly created nodes. Eventually we’ve adapted all corresponding grammar rules to be capable of dealing with the comments. These changes didn’t need to change syntax rules, but only action code that actually created the AST nodes.

This solution worked fine and was very robust concerning unexpected comment occurrences. Unfortunately the treatment of comments in scanner and parser had a devastating effect to the performance of JRuby. Despite being able to improve this flaw it never worked fast enough to be really considered a viable solution.

5.5 Assigning Comments after AST Creation

For our third and last approach we cared about the performance issue occurred in the previous implementation changing JRuby. We wanted a fast assignment or at least to have it as an optional feature to let the user decide if he cares about the comments or not. From the second approach we profited of the comment node creation in the scanner, which was kept in a slightly modified way. So all the comments are collected in a list which is stored in the AST and thus available for everybody who needs them.

Based on the list of comments and the corresponding AST, we implemented simple comment association rules in an AST visitor to associate comments to a node based on their position information. The rules are similar to those which are used in the CDT comment association.

6. Similarities to JDT

As mentioned before, most AST implementations various languages normally don’t contain nodes for representing comments. The Java Development Tools (JDT, the Eclipse Java IDE) provide an extensive range of refactorings for Java. In their refactorings they do process comments, meaning that comments, for example, move along with the according code. Now the big question is if JDT’s comments are represented directly in their AST or if they have to handle the comments separately.

Well, to make it short, they don’t have comments in their AST. The way JDT handles comments is actually not that different from ours. While building the AST, comments get collected and stored alongside with the AST. Then they use a comment mapper to link the comments to real AST nodes. A comment is either assigned before or after a node. If a refactoring modifies an AST node, instead of using its original source positions, it uses extended source positions. These positions are calculated by the comment mapper. If the affected node has comments linked to itself, the positions returned will be the node’s normal positions extended by the position of its linked comments. So if Java code gets moved, deleted or inserted, all the relevant comments will get along automatically.

The difference to our way of comment handling is, in JDT there is no explicit difference between trailing and freestanding comments, they’re both considered to be trailing. JDT furthermore doesn’t link all comments to nodes. A comment in an empty method body for example isn’t linked at all. These differences in itself aren’t disadvantages for JDT, as one might assume. Since JDT effectively works on the original source code the difference if a node is trailing or freestanding is irrelevant because it just moves along with any change. There are possibilities where this approach messes up with some certain comments (for example Extract Method). We on the other hand recreate parts of the AST (build new little ASTs) and use a code rewriter to generate new source code. That’s the place where we need to know if a comment is trailing or freestanding. Like this we’re never messing up with comments. Although in certain cases we lose some of the users formatting of the source code. The good news its that we have a concept which helps us to prevent this.

6.1 Conclusion

We’ve shown in this article how to associate all source code comments with AST nodes. This association enables regeneration of correct code from a transformed AST when refactoring. Our refactoring plug-ins are including all original comments in transformed code most of the time in a position expected by a developer.

Our failures or sub-optimal attempts for collecting comments while parsing and associating them with AST nodes later on, together with the working solutions, can be used as a guideline for others implementing refactoring for their IDEs. For Eclipse-based IDEs, we and our students try hard to extend refactoring support for many languages, among them are C++, Ruby, PHP, Pyhton, JavaScript, and Groovy.
We figure, that this paper enables others as well to implement comment-preserving refactoring tools for more languages and IDEs.

Together with automated unit testing, refactoring is a cornerstone of modern and agile software development practice. We hope, our work enables more programmers to effectively refactor and thus simplify and improve their code. Every programmer, language and IDE deserves good refactoring support or they might become endangered.

6.2 Acknowledgements
We like to thank our headmaster Prof. Hermann Mettler for the support of our work. In addition the authors appreciate the dedication of the hard-working (former) students who implemented Eclipse refactoring plug-ins in the past years and whose results lead to the experiences reported here.

7. Bibliography
* Martin Fowler, Refactoring - Improving the Design of Existing Code, Addison-Wesley, 1999
* CDT Refactoring Update Site, http: //ifs.hsr.ch/ctdrefactoring/updatesite/
* IFS Projects Site, http: //www.ifs.hsr.ch/Projects
* JRuby Project, http: //jruby.codehaus.org/
* Groovy Project, http: //groovy.codehaus.org/
* Eclipse Groovy Project, http: //groovy.codehaus.org/Eclipse+Plugin
* Groovy Refactoring Project, http: //sifsstud4.hsr.ch/trac/GroovyRefactoring
* Student Refactoring Projects, http: //www.ifs.hsr.ch/Projects/Overview/Student_Projects
* dynamic proxy class, http: //java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

References