Journal of Quality and Technology Management
Volume X, Issue II, December 2014, Page 139 – 154

TAXONOMY AND DESIGN CONSIDERATIONS FOR COMMENTS IN PROGRAMMING LANGUAGES: A QUALITY PERSPECTIVE

***M.S. Farooq*[1,2], *S.A. Khan*[2], *K. Abid*[3], *F. Ahmad*[4], *M.A. Naeem*[3], *M. Shafiq*[3a], **A. Abid*[1]**
[1]Department of Computer Science, University of Management and Technology, Lahore.
[2]Abdul Wali Khan University, Mardan.
[3]Department of Electrical Engineering, University of the Punjab, Lahore.
[3a]Institute of Quality & Technology Management, University of the Punjab, Lahore.
[4]Faculty of Information Technology, University of Central Punjab, Lahore.

ABSTRACT

Comments have an important role in software development. Especially medium to large scale projects have a reasonably large code base. Useful and good quality comments play a significant part while maintaining and evolving such projects. In this work we present a taxonomy of comments based on their styles, parsing rules, recursivity, and usage. We also present quality design considerations which the programming languages should ensure so that the support of comments should be free of any side effects.

Keywords: *Comments; programming languages; block comments; inline comments*

1) INTRODUCTION

Comments are most important lexical constructs of programming languages. (Raskin, 2005) Comments are used in the program for various different useful reasons such as, elaboration of programming logic, debugging code, code versioning, copyrights, and documentation purpose. (Hirata and Mizuno, 2011) In most of the software industry, it is mandatory to add comments in the program using special format because they want to create proper program docs (documentation). It highly demanded by quality coding standards to add and write a proper comment in APIs (Application Programmer Interface). (Jiang and Hassan,

2006) (Farooq et al., 2014) (MISRA♦C++, 2008) Comments also plays major role in program readability, maintenance and safety. (Ying et al., 2005) (Fluri et al., 2007) (Farooq et al., 2012) In software industry 90% of the projects belongs to the maintenance and in enhancement phase. (Seacord et al., 2003) (Bacchelli and Bird, 2013) Due to heavy turnover in the software industry most of the time the Programmer who wrote the program are not available when the same program in maintenance and enhancement stage, therefore in this situation a proper commented program helps the maintenance programmer in order to understand programming logic. In APIs, program documents generated based on comments helps a lot for novices and experts to understand language interfaces and function signatures. In this research, we have three contributions, i) highlight major design issues of comments; ii) evaluate comments of seven mainstream programming language and highlight their drawbacks and usage in programs; iii) propose quality and safe comments if any available in these languages.

2) RELATED WORK

Hirata et al. (Hirata and Mizuno, 2011) answer the following question "Do comments describe the code adequately", they have analyzed Integrated Development Environments (IDE) NetBeans and Eclipse, which support Java language. They conclude that end of line comments is more adequate as compared to block comments. Another study focused on finding relationship between comments in source code and bugs ratio. (Lind, 1989) (Tan et al., 2007)

Haouari et al. (Haouari et al., 2011) conduct an empirical study on different open source Java projects, study comments from both point of view i.e. quantitative and qualitative. They also propose a taxonomy of comments, and found that an important portion of comments is dedicated to the communication between programmers or to note for future changes. Tan et al. (Tan et al., 2007) presents a method analyze code automatically and detect comment inconsistencies with source code. In (DePasquale and Locasto, 2010) discussed and presents enhancing quality of source code using comments. Sridhara et al. (Sridhara et al., 2010) presents a novel technique for generating comments for Java methods. In (Mason, 2003) focus on the importance of comments and strongly encourage instructors to motivate their students about commenting in source code. Farooq et al. (Farooq et al., 2014) (Farooq et al., 2012)

presents comprehensive framework for evaluating first programming languages and propose comments as a major quality attribute for novice programmers. They rate a language higher if some programming language supports newline, documentation and mega comment.

3) TAXONOMY OF COMMENTS

In this section, we discuss a taxonomy of comments which is shown in Figure 1. We have discussed comments used in programming languages from the perspectives of styles, parsing rules, recursivity, and usage.

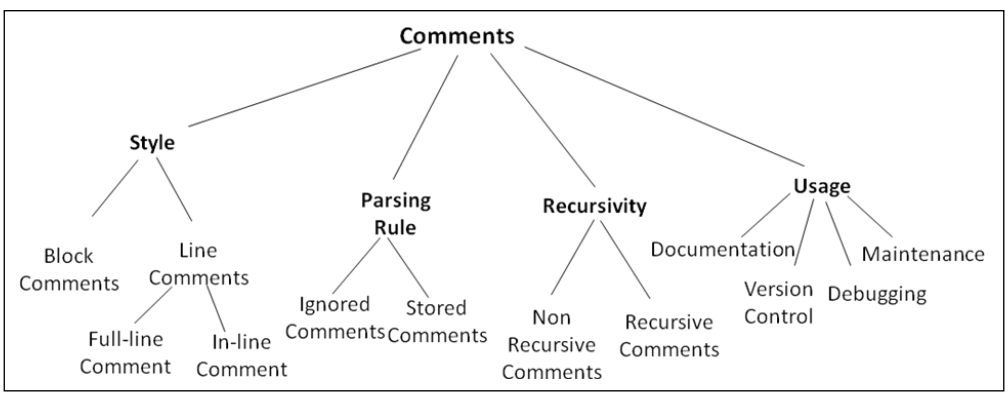

Figure 1: Taxonomy of Comments

3.1) Comment Style

Here, comment style means physical layout (syntax) of the comment. There are three types of comments that falls in style class; i) block comments ii) line comments. The block comment consists of one or more lines, whereas, a line comment is written on a single line. The line comments can be further divided into two types of comments which are full-line comments, and in-line comments.

3.1.1) Block Comments

Block comments can be written on a single line, as well as on multiple lines. These comments have special start and end delimiters, and they can be written anywhere in the program. Coding standards discourage *block comments* as they are prone to errors. (MISRA♦C++, 2008) One serious problem in block comment is that novice programmer may accidentally

forget to end the comment which may swallow useful code. It will be swallowed as shown in code listing 1 (line 1-4). Due to such possible side effects some languages also discourage the usage of block comments, e.g. Ada and Python do not have block comments.

Code Listing 1: Accidental Code Swallowing while using Block Comments

```
1)  /* Set  value of count to 1
2)  count =10;
3)  size= count +1;
4)  /* size should be count plus one */
5)  size =size *count;
```

3.1.2) Line Comments

In-line Comment: In-line comment is an easy and flexible type of comment, can be written anywhere in program delimited with line feed. Quality Coding Standard prefers this type of comment, and also considered to be less error prone for novice programmers. (MISRA♦C++, 2008) Among different types of comments the in-line Comment is the most unambiguous and preferable comment. (Farooq et al., 2014) Code Listing 2 (line 4) and Code listing 3 (line 5) show in-line comment in Java and Ada, respectively. Table 1 shows *in-line comment* supported by mainstream programming languages.

Code Listing 2: In-line comment in Java

```
1)  class Test{
2)  public static void main (int arg []){
3)  System.out.print("Hello World!");
4)  // My First program in Java
5)  }
6)  }
```

Code Listing 3: Inline Comment in Ada

```
1)  with Text_To; use Text_To
2)  procedure MyHello is
3)  begin
4)  put("Hello World!");
5)  -- my first program in Ada
6)  end MyHello;
```

Table 1: Inline Comments Support by Mainstream Programming Languages.

Language	Syntax	Example
Fortran 95	Sign of exclamation!	! my first comment
Ada, SQL, Haskell, AppleScript	Double hyphenation --	--My first comment
C++, Java, C#, PHP, JavaScript, Objective C, Scala, Action Script	Double forward slash //	// my first comment
Python, Ruby, Perl, Unix Shells, Windows power Shell	Number Sign #	# my first comment
VBScript, visual basic,	Single quote ‘	‘ my first comment
Lisp, Scheme	Semicolon ;	; my first comment

Full Line Comment: Full line comments cover a complete line for presenting a comment, where no piece of code is written on the same line. Full Line Comments were used in the early days of programming when programs were written on punch cards and computer used to work with very low memory. For this reason early languages such as Fortran, COBOL and BASIC language support full line comment with various different syntaxes as shown in Table 2. Modern languages also expose inline comments in the form of full-line comment in case where the inline comment commences at the start of the line.

Table 2: Full-line Comments Support by Different Languages

Language	Syntax	Example
Fortran	Character C in position 1	C my first comment
COBOL	Character * at position 7	010111* my first comment
BASIC, batch files	String “REM” at start of line	REM my first comment

3.2) Classification Based on Parsing Rules

It has been a common practice in most of the language specifications and compilers that they ignore comments. However, in other cases the compilers store the comments or interpolate them. In this subsection we have discussed these different parsing behaviors of the compilers while processing the comments.

3.2.1) Ignored Comments

Here ignore means the compiler should discard comment after the first phase (Lexical Analysis) of compilation process. The lexical analyzer checks the syntax of the comment through predefined patterns and then throws away string of comment without generating token for syntax analyzer. C, C++, Java and other C-family languages support this concept.

3.2.2) Stored Comments

Stored comments are important and are highly demanded feature in the programming languages where third party APIs are heavily used. (Sommerlad et al., 2008) (Arafat and Riehle, 2009) (Norick et al., 2010) The stored comments are not stripped-out from target code generated by the compiler, but they are retained throughout the run time of the program in memory. The programmer can retrieve the comments on demand that are synchronized with code. This is possible because the compiler preserves these types of comments into a separate file synchronized with the target code. These types of comments can be retrieved at any time when required to get help regarding interfaces, function, or classes after compilation, as well as during execution. For example Python documentation comment is stored in *.pyc* byte code file. The programmer can retrieve these comments using *moduleName.__doc__* explicit call. The Code Listing 4 defines a function *square*, which can be called with shell session presented in Code Listing 5. It has print result plus documentation comment of function square that are stored in memory. These types of comments are also called *Docstring*. Programming languages like Clojure, Elixir, Lisp and Python support stored (Docstring) comments.

Code Listing 4: *Store in Memory* Comment in Python

```
1) def square(number):
2) """computes square of math.square."""
3) return number * number
```

Code Listing 5 : Python Shell Session Commands

```
1) $ python –c 'print math.square(2);
2) print math.__doc__'
3) 4
4) computes square of math.square
```

3.2.3) Interpolated Comment

A language that supports interpolated string may support interpolated comment. String interpolation is a way to construct a new String value from a mix of constants, variables, literals, and expressions by including their values inside a string literal. Interpolated comment is a special type of comment that are based on interpolated strings and comment strings generated using pre-execution of programs.

3.3) Recursivity of Comments

Nesting of comment inside another comment is considered recursive comment. In this case, the inner comment is redundant as already commented code has been again commented with higher scope comments.

3.3.1 Recursive Comments

Nested comments are used to comment code that is already commented. The Code listing 6 (line 4 to 6) shows the comment nesting in Java. It enhances the language comment defining flexibility, but it has disadvantages too. Recursivity with block comments results in code swallowing problem as discussed in Code listing 2. Here swallowing means accidently some lines are commented out due to missing end part of a block comment. These types of practices are discouraged in quality coding standards and in literature. Rule no 2-7-1 to 2-7-3 of MISRA C++ 2008 (MISRA♦C++, 2008) standard discourages these type of practices. If the compiler checks for accidental code swallowing then recursive comments are not prone to errors.

Code Listing 6: Nested Comments in Java

```
1) Class Test{
2) Public static void main(String a[]){
3) int i=10;
4) /* declare variable i with initial value 10
5) /* print the value of i on console */
6) */
7) System.out.println("valueof i="+i);
8) }
```

3.3.2 Non Recursive Comments

Some languages do not allow nested comments. e.g. In XHTML after starting "<!—" compiler waits for closing "- - >", but if another staring tag "<!—" found before ending compiler will generate error. In such case we cannot write the recursive code.

3.4) Classification of Comments Based on Usage

Comments also facilitate programmer's community to automatic generation of documentation, version control, help debugging of programs, and also support maintenance of the code.

3.4.1) Documentation Comments

Technical documentation is a mandatory requirement of all software's especially usage of (Application Programmer Interface) APIs. (Kramer, 1999) Programmers are encouraged to write documentation comments with the code as these types of comments have been quite useful for both developers who use these APIs, and also for maintenance programmers who change these APIs. Some languages also support documentation compiler that is used to create language documentation from these comments given by the programmer into known a format such as pdf, or HTML. For example Java support ***javadoc*** as documentation generation compiler used to create HTML documentation form documentation comments in source code. C# and VisualBasic.Net support XML documentation Comments. Table 3 show documentation supported in contemporary languages along with automatic documentation tools.

Table 3: Documentation Comments Supported in Contemporary Languages

Language	**Syntax**	**Example**	**Documentation Generation Tool**
Java	/** * */	/** *Print count */	javadoc
Python	"""documentation"""	"""Print count"""	PythonDoc
PHP	/** documentation */	/** Print count */	PHPDoc
C#	/// documentation	/// <print> /// this function prints count /// </print>	Visual Studio.Net
VisualBasic.Net	'''documentation	''' <print> Print count </print>	Visual Studio.Net
Perl, Ruby	# documentation	# Print count	RDoc

3.4.2) Version Control

Sometimes latest version of parsers supports new features that are not supported by older version parsers. This appears to be a most common problem in web browsers, especially in JavaScript parsers. Older browsers do not support Java script code so in order to avoid displaying code in browser, HTML-style comment is used to hide JavaScript code (Code Listing 7). Here old browsers treat JavaScript code as a long HTML comment, and new JavaScript browsers interpret code between <script> and </script> tag.

Code Listing 7: Hiding JavaScript code from script

```
<script type="text/javascript" language="JavaScript">
<!—
function add ( x, y ) { return x+y;}
//-->
</script>
```

Markup languages such as HTML also support these types of comments called conditional comments. This type of comments are used to hide or provide source code from internet explorer. HTML support two types of conditional comments: i) downlevel-hidden conditional (Code listings 8 and 9); ii) downlevel-revealed conditional (Code Listing 10). The first type of comment hides content from browsers that do not support conditional comments. If the result of the conditional expression is true, then code inside a comment is parsed and executed by the browser. Second type enables content in those browsers that do not support conditional comments (code listing 10).

Code Listing 8: downlevel-hidden HTML Comment

```
<!--[if IE 9]>
<h1> Hello Internet Explorer 9.</h1>
<![endif]-->
```

Code Listing 9: downlevel-hidden comment for JavaScript

```
<!--[if gte IE 9]>
<script>
alert("You are running Internet Explorer
9 or a later ");
</script>
<p>Thank you for choosing Internet
Explorer.</p>
<![endif]-->
```

Code Listing 10: Downlevel-revealed conditional comment

```
<![if lt IE 8]>
<h1>Upgrade to latest version of Internet Explorer.</h1>
<![endif]>
```

3.4.3) Debugging

Comments are also used for debugging purpose, which is an essential activity performed by the programmers while detecting and correcting bug in the code. Debugging comments help in conditionally compiling the code while there is a major requirement of observing the behavior of code by watching some variable or object state. For example, C++ supports *mega* comment for tagging, block of code written for debugging purpose by using preprocessor directive (#if ….. #endif). The code listings 11 and 12 present examples of debugging code using two alternatives, first without debugging comments and second by using debugging comment. Clearly, code listing 12 is a better choice than code listing 11 because it conserves memory space used for the counter variable. Compiler conditionally compiles code, if _debuFlag has been defined by the programmer using #define _debugFlag preprocessor directive. C# supports these types of comments.

Code Listing 11: Counting Number of nodes in a Linked List for debugging without mega comment in C++

```
Bool debugFlag=flase;
int count=0;
void traverse(node *p){
……
……
If(debugFlag) count ++;
}
```

Code Listing 12: Counting Number of nodes in a Link List for debugging with mega comment in C++

```
#ifdef _debugFlag
int count=0;
#endif
void traverse(node *p){
……
……
#ifdef _debugFlag
count ++;
#endif
}
```

Extensible Markup Language (XML) also supports mega comment by writing DTD (Document Type Definition) in ***IGNORE*** block (Code Listing 13).

Code Listing 13: Mega Comment support in XML

```
<![IGNORE[
<!ELEMENT student (name, department)>
<!ELEMENT name       (#CDATA)>
<!ELEMENT department   (#CDATA)>
]]>
```

3.4.4) Maintenance:

Comments in programming languages support maintenance of software which is an essential activity that supports rectification of errors, and also helps evolving the software system. Furthermore, it is not necessary that the same person would carry a project's development for the whole life time of the project, and would also be responsible for its maintenance. Rather, usually there is a new person every time the project is evolved. To this end, programmers write good quality comments that explain the logic and salient points that are considered while coding. Such comments help others understand a piece of code, and help detecting the errors from the code.

4) DESIGNING GOOD QUALITY COMMENTS

In section 3, we have discussed the comments in programming languages from various different perspectives. We have seen that there are some issues in using certain types of comments e.g. nesting of block comments poses the problem of code swallowing. Hence, it is important to define quality standards for designing comments in programming languages. Such quality standards should ensure that there should not be any side effects while using any type of comment. The general consideration for a good quality comment are that it should not compromise on program's readability, writeability or on safety issues.

The syntax of the comments is covered in the lexical specification of the compiler of a language. The lexical analyzer checks comment syntax as other lexical units (identifiers, operators etc.), but a general difference is that no token is generated for comments, such that the syntax analyzer does not get anything related to comments from the lexical analyzer. The comment design is a major lexical issue in programming language. A language designer should incorporate the following design

considerations while defining different variants of comments in the proposed programming language:

i) What is the starting delimiter or string for comments?
ii) What is the ending delimiter or string for comments?
iii) Does the starting position matter in comments formatting?
iv) Is comment nesting allowed?
v) Does not have clash with other language construct

4.1) Comment Starting Delimiter/String

Starting delimiter should be an important consideration while designing comments. Generally, it is recommended to have one, two or more starting characters. Early programming languages have been using one character as starting character, but it may create ambiguity when programmer accidently inserts that character from the keyboard. Two characters start is a reasonably good choice because it minimizes the issue or ambiguity of accidental insertion. Three or more characters as starting string is too redundant and is creating writeablility problems for comments

4.2) Comment Ending Delimiter/String

Ending delimiter or string also plays a major role in program readability and writeability. It depends on the comments nature either end of line, Full line or block based. Full line and end of line comments use line feed character as ending delimiter, while block comment uses some special character or set of characters. The considerations for more than one character should be the same as of starting delimiter.

4.3) Starting Position of Comment

Form which position we start commenting on the code has not a major issue in current languages, but in the early era of programming languages when punch cards were used, comment starting position used to carry significant weightage. Fortran, COBOL and BASIC use fixed starting position for comments, and each comment occupies whole line. So, these languages do not expose inline comments.

As a matter of fact, the starting position of a comment or placement of a comment is dependent upon the size of the comment. For instance, for small comments we use inline comments, whereas for large comments such as some comments about explaining the functionality of a method are presented before the definition of a method.

4.4) Comments Nesting

Comments within comments create serious readability problem. Such type of comments are strongly discouraged by quality coding standards. (Weinman, 1983) (MISRA♦C++, 2008) They also creates classes between comments format if language support more than one type of comments. For example, if we nest new line comment with in block comments, it generates ambiguous results. i.e. newline comment dominates block comment ending or block comment dominates newline comment. It also creates swallowing code accidently due to wrong nesting. Some programming languages do not allow this due to expected misuse of nesting in the comments. Some compilers generate warning on nested comments. Most of pretty printers used in language IDEs also accidently marks code green (normally green color is used for comments) in case of misuse of nested comments. Quality coding standards strongly discourage nesting comments and such comments are also not practiced in the software industry.

4.5) Does not Clash with other Language Constructs

A good quality design of comments for a programming language should ensure that the symbols used as starting or ending delimiter should not have any syntactic clash with any other language construct (Benepe, 1984) e.g. operators, keywords etc. For instance the Code Listing 14, shows a conflict with the syntax of a comment in the syntax of C++ where the division operator is followed by an '*' which is meant for pointer. In order to avoid this issue, C++ programmer is bound to write extra spaces between code (Code Listing 15). Such issues should be addressed by the languages to make good quality comments.

Code Listing 14: Pointer Syntax clash with Block Comments in C++	**Code Listing 15**: Significance of space in C++	
a =15/*ptr + 5;	**Incorrect (no space)**	**Correct (with space)**
	a=15/*ptr + 5;	a =15/ *ptr + 5;

5) CONCLUSION

In this article, we have presented a taxonomy of comments used and supported by programming languages. We have highlighted the importance, usage, types, and issues while discussing the comments in detail. We have also presented quality design considerations for incorporating comments into a programming language's design. This would certainly help minimizing the possible side effects of using comments in programming languages.

In future, we intend to extend this work by performing a rigorous analysis of contemporary programming languages by analyzing their conformance to the proposed design guidelines.

REFERENCES

Arafat, O. & Riehle, D. 2009. The commenting practice of open source. *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications.* Orlando, Florida, USA: ACM.

Bacchelli, A. & Bird, C. 2013. Expectations, outcomes, and challenges of modern code review. *Proceedings of the 2013 International Conference on Software Engineering.* San Francisco, CA, USA: IEEE Press.

Benepe, D. B. 1984. In defense of simplicity of comment syntax. *SIGPLAN Not.,* 19**,** 32-33.

Depasquale, P.J. & Locasto, M. E. 2010. Teaching students effective practices for commenting computer source code: tutorial presentation. *J. Comput. Sci. Coll.,* 25**,** 53-53.

Farooq, M.S., Khan, S. A. & Abid, A. 2012. A Framework for the Assessment of a First Programming Language. *Journal of Basic and Applied Scientific Research,* 2**,** 8144-8149.

Farooq, M.S., Khan, S.A., Ahmad, F., Islam, S. & Abid, A. 2014. An Evaluation Framework and Comparative Analysis of the Widely Used First Programming Languages. *PLoS ONE,* 9**,** e88941.

Fluri, B., Wursch, M. & Gall, H.C. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on, 28-31 Oct. 2007 2007. 70-79.

Haouari, D., Sahraoui, H. & Langlais, P. How Good is Your Comment? A Study of Comments in Java Programs. Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on, 22-23 Sept. 2011 2011. 137-146.

Hirata, Y. & Mizuno, O. 2011. Do comments explain codes adequately?: investigation by text filtering. *Proceedings of the 8th Working Conference on Mining Software Repositories.* Waikiki, Honolulu, HI, USA: ACM.

Jiang, Z.M. & Hassan, A. E. 2006. Examining the evolution of code comments in PostgreSQL. *Proceedings of the 2006 international workshop on Mining software repositories.* Shanghai, China: ACM.

Kramer, D. 1999. API documentation from source code comments: a case study of Javadoc. *Proceedings of the 17th annual international conference on Computer documentation.* New Orleans, Louisiana, USA: ACM.

Lind, R.K. 1989. An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort. *IEEE Transactions on Software Engineering,* 15**,** 649-653.

Mason, J. 2003. Comments considered harmful. *SIGCSE Bull.,* 35**,** 120-122.

Misra♦C++ 2008. *Guidelines for the use of the C++ language in critical systems*.

Norick, B., Krohn, J., Howard, E., Welna, B. & Izurieta, C. 2010. Effects of the number of developers on code quality in open source software: a case study. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement.* Bolzano-Bozen, Italy: ACM.

Raskin, J. 2005. Comments are More Important than Code. *Queue,* 3**,** 64-65.

Seacord, R.C., Plakosh, D. & Lewis, G.A. 2003. *Modernizing legacy systems: software technologies, engineering processes, and business practices*, Addison-Wesley Professional.

Sommerlad, P., Zgraggen, G., Corbat, T. & Felber, L. 2008. Retaining comments when refactoring code. *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications.* Nashville, TN, USA: ACM.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L. & Vijay-Shanker, K. 2010. Towards automatically generating summary comments for Java methods. *Proceedings of the IEEE/ACM international conference on Automated software engineering.* Antwerp, Belgium: ACM.

Tan, L., Yuan, D., Krishna, G. & Zhou, Y. 2007. /*icomment: bugs or bad comments?*. *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles.* Stevenson, Washington, USA: ACM.

Weinman, J.B. 1983. Nestable bracketed comments. *SIGPLAN Not.,* 18**,** 44-47.

Ying, A.T.T., Wright, J.L. & Abrams, S. 2005. An exploration of how comments are used for marking related code fragments. *SIGSOFT Softw. Eng. Notes,* 30**,** 1-4.