

# Automatic Documentation Generation

*Duane Strong*

*duanes@strongenging.com*

*1/6/2005*

## Introduction

This paper is an investigation into the current state of tools that allow for automatically generating code documentation from source code.

## Background

Tools have existed for a number of years that extract information from source code and format it for code documentation purposes generating HTML, RTF, or other types of output. Most UML modeling tools such as Rational Rose, Together, and Enterprise Architect support this feature, as well as tools made specifically for this purpose such as Doxygen, DocJet, and CodePrint Valet There are many others ( see <http://www.stack.nl/~dimitri/doxygen/links.html> ) . The creators of the Java language considered this functionality important enough to include the Javadoc tool with the distribution of the language. Since that time many tools have settled on the Javadoc style of comment tags to support comment extraction. Using the Javadoc conventions in the source code will provide the most flexibility between documentation tools.

## Javadoc

The Javadoc tool <http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javadoc.html> uses the structure of the Java language in addition to special comment styles to extract information about the source code. It uses this information to format HTML pages that display source code documentation.

You can include *documentation comments* ("doc comments") in the source code, ahead of declarations for any class, interface, method, constructor, or field. A doc comment consists of the characters between the characters `/**` that begin the comment and the characters `*/` that end it.

```
/**
 * This is the typical format of a simple documentation comment
 * that spans two lines.
 */
```

The comment text is then associated with its following Java declaration item and placed into the output with the documentation of that item.

Inside the doc comment following the text is the tag area. This tag area may contain block tags. A block tag begins with the `@` sign (after any white space or `*` characters) and indicates that the following keyword marks the subsequent text with special semantics.

The more useful blocks tags are:

- `@author name-text` marks the *name-text* as the author entry
- `@param param-name description` associates the *description* with the *param-name* for a method
- `@return description` associates the *description* with the method return value
- `@see link` creates a "see also:" html link entry to other websites or an internal cross reference to other class methods
- `@version version-text` associates *version-text* with the version of the document

Another type of tag called the in-line tag can appear anywhere in a doc-comment. They are like block tags but must be enclosed with curly braces. Useful in-line tags are:

- `{@link}` inserts a hyperlink to another class or method in-line
- `{@linkplain}` inserts a hyperlink in normal text font

## Doxygen

Doxygen <http://www.stack.nl/~dimitri/doxygen/index.html> is a popular open source tool for generating documentation from source. It supports Java, C++, C, and other languages. It supports the Javadoc style of source comments, as well as many other styles via a configuration file. Notably it supports a form of comment that can be placed beside or after the declaration instead of only before as in Javadoc. It has many extended features

and can generate output in HTML, RTF, and LaTeX. Simple class hierarchy diagrams are also generated.

## Enterprise Architect

Like most modeling tools, Enterprise Architect <http://www.sparxsystems.com.au/> can extract information from Java, C++, C, and other languages and use additional model information to generate documentation. Strictly speaking the documentation is generated from the model data but the model data can be created via reversing source code. It can output HTML or RTF files. It supports Javadoc style comments but is configurable for input and has a configurable output designer to tweak the output to particular tastes. While not as configurable as Doxygen, it does include UML class diagrams and other information that the model contains beyond what can be gathered from source.

## Necessary Code Comments

No comments are necessary to generate documentation with these tools as most of the information gathered comes from the structure of the language itself. However by supplying doc comments or by simply changing existing comment styles additional information is extracted and placed into the documentation.

### Example code fragment without block comments

```
class DrvrUart : public Driver {
public:

    /* default method is supplied that returns next character in the receive queue
    returns next character in the receive queue (cast to long) or QUEUE_IS_EMPTY */
    virtual long int nReadByte( );

    /* default read method is supplied that empties the receive queue
    pBufferPtr points to buffer to fill (should be at least the queue size)
    iNumBytesReadPtr points to returned integer count of bytes read
    returns ok or error */
    virtual tError xReadBytes( unsigned char * ucBufferPtr, int * iNumBytesReadPtr );
```

Doxygen RTF output fragment

long int DrvrUart::nReadByte () [virtual]

virtual [tError](#) DrvrUart::xReadBytes (unsigned char \* *ucBufferPtr*, int \* *iNumBytesReadPtr*)  
[virtual]

## Example code fragment with additional block comments

```
/** This abstract class defines an interface to a UART device.
    Hardware specific classes are derived from this class */
class DrvrUart : public Driver {

public:

    /** default method is supplied that returns next character in the receive queue
    @return next character in the receive queue (cast to long) or QUEUE_IS_EMPTY */
    virtual long int nReadByte( );

    /** default read method is supplied that empties the receive queue
    @param ucBufferPtr pointer to buffer to fill (should be at least the queuesize)
    @param iNumBytesReadPtr pointer to returned integer count of bytes read
    @return returns ok or error */
    virtual tError xReadBytes( unsigned char * ucBufferPtr, int * iNumBytesReadPtr );
```

## Doxygen RTF output fragment

### **long int DrvrUart::nReadByte () [virtual]**

default method is supplied that returns next character in the receive queue

#### **Returns:**

next character in the receive queue (cast to long) or QUEUE\_IS\_EMPTY

### **tError DrvrUart::xReadBytes (unsigned char \* ucBufferPtr, int \* iNumBytesReadPtr) [virtual]**

default read method is supplied that empties the receive queue

#### **Parameters:**

*ucBufferPtr* pointer to buffer to fill (should be at least the queuesize)

*iNumBytesReadPtr* pointer to returned integer count of bytes read

#### **Returns:**

returns ok or error

## Sample project output

The example driver code was put through both Doxygen and EA Architect. Only minor effort was put forth to tailor the output by adjusting settings.

Doxygen RTF [doxygenrtf\refman.rtf](#)

Doxygen HTML [doxygenhtml\index.html](#)

EA Architect RTF [eaarch.RTF](#)

EA Architect HTML [eaarchhtml\index.htm](#)