For Macintosh Programmers & Developers
MacTech MAGAZINE

DEVELOPERS TOOLS

By Aaron Montgomery

Documenting your code

Generating code documentation with doxygen on Mac OS X

About the author...
Aaron teaches mathematics at Central Washington University for a living and enjoys mountain biking in the summer and brewing beer in the winter. You can reach him at monsterworks@mac.com.

INTRODUCTION

One of the nice features of Mac OS X is that it opens up access to a large number of developer tools that used to be restricted to UNIX and Windows. One such tool is `doxygen`, an application that aids in code documentation in a manner similar to `javadoc`. This article describes how to set up `doxygen` on Mac OS X and assumes that you have only limited UNIX experience. As this article developed, it became more of an introduction to UNIX basics inside the `Terminal` and less an introduction to `doxygen` than I originally planned. This is due both to the fact that `doxygen` already contains a manual and numerous examples (that I found useful), as well as the intended audience (Mac programmers new to UNIX). The article describes how to install `doxygen` on Mac OS X; how to run `doxygen` from the `Terminal`; how to run `doxygen` as part of `Project Builder`'s build command; and, finally, how to extend `doxygen` to provide diagrams with an application called `dot`.

The first step to using `doxygen` is getting a copy of the program. There is a binary executable available at the `doxygen` website listed below (version 1.3.4 at the time I finished this article). Download the file, decompress it and place the resulting folder somewhere. I placed mine in the `Applications` folder.

WELCOME TO THE SHELL

Read the following paragraph:

You will need to install the `doxygen` binary along your shell's path. You can then test your installation by changing to the `doxygen/examples/` directory, removing the directory called `afterdoc`, then executing the command `doxygen afterdoc.cfg` (which should regenerate the directory). You can browse the generated documentation by opening the file `index.html` inside the `afterdoc/html/` directory.

Got it? If so, you can probably just skim the installation section of this article. If only some (or even none) of the paragraph made sense to you, please keep reading while I try to lead you through the steps carefully in the rest of this section. What I cannot do is give you a full introduction to the UNIX shell (I am still learning it myself). If that is your goal, the most important UNIX command is `man`. I used it to confirm that I was correct when I described each command below. You have probably read it before in MacTech, but I will write it again: use `man` frequently as you explore the shell.

Start up `Terminal` (in `Applications/Utilities`) and you are now in a shell. If you are reading this section, I will assume that you are working in the tcsh shell without much modification (for example, the default settings from Apple). Below is a description of how to determine if that assumption is accurate. You should type in the command following the `%` symbol and the computer should respond with something similar to the remaining lines. Your prompt might be longer than just a single `%` (probably indicating the computer name and your user name), but I removed that information from the examples below. I will use `[return]` to indicate pressing the return key, and, similarly, `[tab]` for the tab key.

```
% echo $version[return]
tcsh 6.10.00 (Astron) 2000-11-19 (powerpc-apple-darwin)
options 8b,nls,dl,al,sm,rh,color
```

The command `echo` asks the shell to print the result to the screen and the expression `$version` asks for the value of the variable `version`. If you are not running tcsh, it might seem like the commands `chsh` and `chpass` will allow you to change shells. Well, they don't (at least not on my machine). However, the application `NetInfo Manager` (located in the `Applications/Utilities` folder) will allow you to change your login shell. I am going to assume that you are using tcsh. You can place the `doxygen` executable anywhere, but since you will be executing it from a UNIX shell, you will need to make sure the shell can find the executable. The shell keeps a list of directories that it looks through when you issue a command. This information is stored in the variable `path`. You can discover the directories in this list using the command.

```
% echo $path[return]
/sw/bin /sw/sbin /bin /sbin /usr/bin /usr/sbin
/usr/local/teTeX/bin/powerpc-apple-darwin-current
/usr/local/bin /usr/X11R6/bin
```

If you have not installed anything, your path will be much shorter. Here is some idea of what these directories hold (based on my poking around inside of them): `bin/` directories hold binaries (applications), `sbin/` directories hold system binaries and `usr/` directories hold user files. When I installed `teTeX` (a TeX installation), the `/usr/local/` directories were added. When I installed `fink` (an application for managing UNIX packages), the `/sw/` and `/usr/X11R6/` directories were added. We will discuss how to add directories to your `path` later in the article. For now we will use one of the standard directories that Apple sets up: `/usr/bin/`. If you decide to add `dot` following the instructions I provide below, you will create the `/usr/local/bin/` directory and may want to keep `dot` and `doxygen` together inside this directory (and make appropriate changes below).

You could either move the application into one of these directories using `mv` or copy it using `cp`, or do what I am going to describe below and make a link (the UNIX version of an alias). Personally, I place the application in the `Applications` folder so that it is easy to get to from the `Finder` and then create a link into the appropriate directory along the path.

There are two types of links under UNIX: hard and symbolic. You will almost never use a hard link. Deleting a hard link causes the original file to be deleted, a very non-Mac behavior (and so not something that should be set up lightly). Symbolic links behave like aliases, deleting the link leaves the original file alone. The system stores a symbolic link as a path to the target so replacing one folder in the path with an identically named folder will change the symbolic link's target. This allows you to install upgrades by replacing folders in the `Finder`. In order to facilitate this, I removed the version information from the `doxygen` folder's name.

The following is the command to create a link in the `Terminal`. You will need to be logged in as an administrator (for example, using the account you created when you first installed Mac OS X). Assuming that you are mimicking my set up, here is the command (type it on one line, it is split here to fit in the article).

```
%ln -s /Applications/doxygen/bin/doxygen
/usr/bin/doxygen[return]
ln: /usr/bin/doxygen: Permission denied
```

As a normal user, you are not allowed to touch the `/usr/bin/` directory. The trick is to find a phone booth and change into your super user outfit. This is not as hard as it seems. The command you want to use is the following.

```
%sudo ln -s /Applications/doxygen/bin/doxygen
/usr/bin/doxygen[return]
```

You should be prompted for your password and once it is accepted, the link will be created (assuming you are using an administrator account). Here is a quick breakdown on the command: `sudo` tells the shell that it should execute this command as an administrator (which is why you will need to type in your password). The command `ln` creates a link. The option `-s` creates a symbolic link. The next two arguments are the path to the original file (in `Applications`) and the path to the alias you are creating. You may need to change this (for example, if you placed `doxygen` in the `Developer` folder instead of the `Applications` folder). If either of these paths contains spaces, you will need to put the path in quotes or escape the space with a `\`. Once authenticated, you can use `sudo` for the next 5 minutes without reentering your password. You may also want to make symbolic links for `doxysearch` and `doxytag` in case you decide to use them later.

You can test this installation by typing the command below.

```
%which doxygen[return]
/usr/bin/doxygen
```

The `which` command tells you which file the shell will use when you use the command doxygen. On my computer, the file /usr/bin/doxygen will be used and all is good. But your response may have been the much less satisfying.

```
%which doxygen[return]
doxygen: Command not found.
```

The response here does not look very hopeful. The problem is that the shell only searches the directories in the `path` variable at startup and when the `path` variable is changed. In this case, we have added the `doxygen` command behind the shell's back and need to tell it to recheck the path directories. The command `rehash` does this.

```
%rehash[return]
%which doxygen[return]
/usr/bin/doxygen
```

Next we will use one of the `doxygen` examples to make sure `doxygen` will run. The command `cd` changes directories. We will use command completion to get to the examples directory. Start by typing the following (there is no `[return]`, this is not a misprint, do not type `[return]` yet).

```
%cd /Appl[tab]
```

On my machine (and possibly on yours) there are multiple directories starting `/Appl` (for example, `Applications` and `Applications (Mac OS 9)`). Pressing `[tab]`, completes the path to the longest sequence of common characters and you will need to type in some more characters to indicate which one you want. This is tcsh's command completion feature. If you do not remember what the next character should be, you can type `[^D]` (control-D) to get a list of all the possible completions. As before, if you have spaces in your directory names, you will need to precede them with a `\`, similarly, parentheses need to be escaped. Now continue typing (without hitting `[return]` yet).

```
/dox[tab]ex[tab]
```

Assuming your set-up mimics mine, the command should now read

```
%cd /Applications/doxygen/examples/
```

and pressing `[return]` will execute the command on the line.

You can check the contents of this directory by typing the command

```
%ls | more[return]
```

This `ls` command asks the shell to list the directory's contents, but rather than just scroll it all onto the screen, we send the output to the `more` command using a pipe (`|`). The `more` command will wait for you to press `[space]` before scrolling down a page (`[return]` will scroll down one line and `q` will exit the listing). Next you want to remove the `afterdoc/` directory. The `rm` command is used to remove (delete) things.

```
%rm afterdoc[return]
rm: afterdoc: is a directory
```

No good, plain `rm` only works for files. You will need to use the `-r` option to recursively remove the directory (`rmdir -p` also works, but is more cautious, requiring the directories to be empty). We are now going to use tcsh's history feature to edit the incorrect command. Press the up arrow and the last command you executed will return to the command line (repeated up arrows will work back through older commands and down arrows work forward), now use the right/left arrow keys to get to the spot just after `rm` and add a `-r` between the `rm` and `afterdoc` to get the command below (execute it by pressing `[return]`).

```
%rm -r afterdoc
```

You can confirm that the directory was removed with `ls | more` (either type it in or use the up arrow a few times to return this to the command line and then press `[return]`). The file should be in the first page of listings. Watch out with `ls`, it places all capital letters before all lower case letters, so if you are looking for a file starting with "a", it may actually appear after files starting with "Z". Once you confirm that `afterdoc/` is missing, you can type `q` to exit the listing. Now we are going to use `doxygen` for the first time.

```
%doxygen afterdoc.cfg | more [return]
```

You might want to scan through the messages or type `q` to get back to the command line. You can check that this worked with the following.

```
%cd aft[tab]/h[tab][return]
%open index.html[return]
```

The `cd` command should move you to the `afterdoc/html/` directory and the `open` command opens index.html in your Mac OS X web browser. You could also navigate to this folder in the `Finder` and open the file with a double click.

THE CONFIGURATION FILE

Okay, now it is time to create your own documentation. The first step is to get a template configuration file. When creating the sample project I created a folder called `doxygen` in the same folder as the project file in the `Finder`. I then moved to the new directory (with a `cd` command) and then typed.

```
%doxygen[return]
```

Like many UNIX commands, using the command `doxygen` with the wrong number of arguments provides a brief statement describing how to use the command. Unless you actually have a file called `Doxyfile` in the current directory, you will get a list of uses for `doxygen`. I created a default template file (using the `-g` option).

```
%doxyfile -g Fish.doxycfg[return]
```

I do not use the default name `Doxyfile` because I like to be able to tell the associated project from the configuration file's name and by not naming my file `Doxyfile`, I can use the `doxygen` command to get help. We now need to adjust the file to suit our needs. You can edit the file in a Mac OS X text editor (such as `Project Builder`, `AlphaX` or `BBEdit`) or a UNIX text editor (such as `emacs`, `vi` or `pico`). Trying to describe which choice you should make is a can of worms I will not even try to sort out here. For those who are new to UNIX, I would suggest one of the Mac OS X editors listed above. You will be able to open the file from the `Finder` with a double-click (once you set its Open with… field in the Get Info dialog) or from within the Open dialog of the application. Alternatively, you can use the command below to open the file with `Project Builder` from the shell (assuming `Project Builder` is in the location Apple placed it). The `-a` option allows you to specify the application that will be used to open the file. If you find that you are doing a lot of opening with the same application, check out the section below where I talk about the `.tcshrc` file.

```
%open -a /Developer/Applications/Project\ Builder.app
Fish.doxycfg[return]
```

The template file is heavily commented (unless you used the `-s` option when building it) and is also described in the `doxygen` documentation. I only adjusted the lines shown below for the sample project.

```
PROJECT_NAME  = "About Box"
OUTPUT_DIRECTORY = ./doxygen
EXTRACT_ALL = YES
EXTRACT_PRIVATE = YES
EXTRACT_STATIC = YES
TAB_SIZE = 4
OPTIMIZE_OUTPUT_FOR_C  = YES
INPUT = ./Source
SOURCE_BROWSER = YES
ALPHABETICAL_INDEX = YES
GENERATE_LATEX = NO
```

The `PROJECT_NAME` line sets up a project name that will be used in the documentation and the `TAB_SIZE` line indicates that I use tabs that are four characters wide (since `doxygen` will convert tabs to spaces when formatting the documentation). I also told `doxygen` that all source is C source (in the `OPTIMIZE_OUTPUT_FOR_C` line). The `SOURCE_BROWSER` line tells `doxygen` to include the source of the files as part of the documentation. I also requested an alphabetical index of all of the compound structures used with the `ALPHABETICAL_INDEX` line.

The `OUTPUT_DIRECTORY` line specifies where the output is to be placed. The directory called dot (`.`) refers to the current directory and dot dot (`..`) is the parent directory of dot (these are available in the shell so you can use them there also). To prepare for later work, I will assume that the current directory is the directory containing the project file (and the parent of the directory containing the configuration file). If you plan to execute the `doxygen` command from same directory as the configuration file, then you might want to use dot as the output directory. The `GENERATE_LATEX` line turns off the option to generate LaTeX documentation (the template file creates both HTML and LaTeX documentation, rtf and man documentation are available also but turned off in the template). The `INPUT_DIRECTORY` specifies where the header and source files reside. If you were planning to execute from within the same directory as the configuration file, you will need to change the input directory to `../Source`.

The three lines concerning `EXTRACT`ion are telling `doxygen` that I want everything documented. If I wanted to hide implementation details, I could set these to `NO` and limit the visibility of the implementations in the documentation. Toggling these provide you with control over what is documented and allow you to create multiple sets of documentation, for example, one set for people developing the package (with these set to `YES`) and another for people who are using the package (with these set to `NO` and maybe even the `SOURCE_BROWSER` also set to `NO`). If you want more details about what will be extracted, please consult the `doxygen` manual, however, there is one detail worth repeating here. If you have `EXTRACT_ALL` set to `NO`, definitions within namespaces are only documented if the namespace is documented. Similarly, global definitions will only be documented if the file in which they occur is documented (with a `@file` tag).

A good way to figure out what doxygen can do is to poke around in the configuration file and to toggle some switches and compare the output. Although the configuration file is long (over 1000 lines including comments), the comments help guide you through the various options. You can test the configuration file from the command line by first changing to the directory containing the project file and then issuing the command

```
%doxygen doxygen/Fish.doxycfg[return]
```

This should generate an html folder inside the doxygen folder full of documentation that you can view with your browser.

PREPARING TO DOXYGENATE YOUR CODE

You can run doxygen without using any special comments, however, you get better results by adding them. This section is short because I found the manual and examples included with doxygen to be quite good and suspect that the primary obstacle to using it on Mac OS X is going to be the installation process described above. While reading the examples here (and those provided with doxygen), be aware that doxygen is configurable, and if the comment style below feels unnatural to you, be sure to read the doxygen manual to see if your preferred comment style is supported.

To prepare your code so that doxygen can generate good documentation you will need to place special comment blocks inside your code. There are a wide variety of options using both C and C++ style comments. I tend to use C++ style comments (so that I can use C style comments to eliminate long blocks of code during testing). If you prefer C style comments, please refer to the doxygen manual for the ways they can be used (alone or in conjunction with C++ style comments). When reading a source file, doxygen will recognize comments that begin with either /// or //! as a special comment that contains documentation information. I prefer to use //! because it sticks out when I visually scan a file. You can structure the comments with a limited subset of HTML and by doxygen tags (you can use either @ or \ as the escape character, I use @ because it sticks out when I visually scan a file). I have created a (somewhat silly) project to give you some idea of what doxygen can do. The excerpt below is intended to provide you with a feel for the type of comments that can be added to the source to provide documentation. As usual in magazine printouts of code, watch out for wrapped lines (or better yet, look at the actual source file, available at the MacTech site as well as my website listed below).

excerpt from fish.h

```
//! @brief This structure information about fish in my freezer.
//!
//! It doesn't really do justice to the great variety of things that are fish
//! in my freezer. For instance, it might be nice to know how old they are or
//! whether they still have their heads.
typedef struct
{
  fishTypes myType;  //!<what type of fish we are
describing
  int myWeight;    //!<the weight of the fish
} fish_t;
```

```
//! @brief The number of fish that fit in my freezer.
//!
//! Of course, it really depends on the size of the fish and if
//! I upgrade my freezer I will certainly need to enlarge this number.
//! The only reason for all this typing is create a detailed description.
#define MAX_FISH 10

//! @brief The fish in my freezer.
extern fish_t freezer[MAX_FISH];

//! @brief Fills the freezer with fish.
void FillFreezer();

//! @brief Returns the total weight of all fish of one type in the freezer.
int Weigh(fishTypes inType);
```

excerpt from fish.c

```
//! This function iterates through all the fish in my freezer of
//! a particular type and sums up their weights.
//! This function replaces the defective WeighEm() function.
//! @param inType is the type of fish I want to weigh.
//! @returns the total weight of all fish of that type in my freezer.
//! @bug I can't figure out what's going wrong with this function.
int Weigh(fishTypes inType)
{
  int i, result;
  for (i = 0; i < MAX_FISH; i++)
  {
    if (freezer[i].myType = inType)
    {
      result = result + freezer[i].myWeight;
    }
  }

  return result;
}
```

When processing the code, `doxygen` will look for documentation for `#defines`, `enums`, `structs`, global variables and functions (and other things) and will produce both a brief description as well as a detailed description. As a convention, I use the `@brief` tag for brief comments although `doxygen` has a default algorithm for determining the brief and detailed descriptions if you do not specify them. In the excerpt above, `@param` indicates a parameter to the function, `@returns` describes the return value and a `@bug` tag adds this comment to the bug list (bonus question: can you find the bug?). If the comment contains something that looks like a function call (for example, the `WeighEm()` above), `doxygen` will try to hyperlink it to

the appropriate function's documentation. Another nice feature is that doxygen will try to crosslink everything to anything referred to by it and anything to which it refers. The following is a text rendition of the documentation for the function Weigh(). I have indicated the hyperlinks with underlines.

Sample Documentation

```
int Weigh
(
  fishTypes
  inType
)

Returns the total weight of all fish of one type in the
freezer.

This function iterates through all the fish in my freezer
of a particular type and sums up their weights. This
function replaces the defective WeighEm() function.

Parameters:
  inType
    is the type of fish I want to weigh.

Returns:
  the total weight of all fish of that type in my freezer.

Bug:
  I can't figure out what's going wrong with this function.

Definition at line 38 of file fish.c.

References freezer, MAX_FISH, fish_t::myType, and
fish_t::myWeight.

Referenced by main().
```

You can find more samples in the doxygen examples as well as the project associated with this article (at the MacTech site as well as my website listed below).

ADDING A DOXYGEN STAGE TO YOUR BUILD

I am going to leave further exploration of what doxygen does to the doxygen manual and examples and move on to controlling doxygen from within Project Builder. I will describe how I added a step to create the documentation at the end of the build. You could place this step at other points in the build process or even create a special target that generates documentation and nothing else. I selected the Edit Active Target 'Fish' menu item from the Project menu. In the left pane, I expanded Build Phases (well, not quite, it was already expanded), and added a new build phase here by selecting the ResourceManager Resources

item and then New Shell Script Build Phase in the New Build Phase submenu of the Project menu. There are two places I needed to add information. The first is the Shell: text field. This field is set to `/bin/sh` (the Bourne shell) by default. The Bourne shell is often used for scripts for portability reasons, but I want to use tcsh for this project since our script is a simple single line command and I spend most of my time in tcsh. To do this, I changed the `/bin/sh` to `/bin/tcsh`. The second field is the command to be executed and in this case I typed `doxygen doxygen/Fish.doxycfg`. That wasn't so tough. Now at the end of any successful build, `doxygen` will create a folder full of documentation (and `doxygen`'s messages are sent to the Build window). If you just have a couple of simple commands to execute in the shell during a build, you can add them with separate Shell Script Build Phases. For more complicated steps, you might want to actually write a file that contains the script and then use the `source` command to execute that file from `Project Builder`.

DOXYGEN AND DOT

When creating documentation, `doxygen` is also able to generate diagrams describing various relationships — in particular, header inclusions and class hierarchies. In order to do this, `doxygen` needs to use an application called `dot`. First download `dot` for Mac OS X from the GraphViz website listed below in the references (my copy was version 1.10 built on 08/27/2003). If working with the shell is still relatively new, you should grab the Installer package (which is what the instructions below assume because that is what I did). Download it and install it according to the instructions. I would suggest letting it place everything in the default location and the instructions below assume that you did this. If you place it somewhere else, you will need to do a few more steps (information can be found at the GraphViz website). You could check to see if the files were installed with the following commands.

```
%cd /usr/local/bin[return]
%ls dot[return]
dot
```

When I asked the shell to `ls dot`, I am asking it to list every file called `dot` and if `dot` does not exist you will get an error message. You can use `*` as a wildcard character, for example

```
%ls *dot*[return]
dot   dot2gxl   dotneato-config   dotty   gxl2dot
```

This is more convenient than using `ls | more` in places where you know some part of the filename. Now that you know the file exists, find out if the shell can find the installation with which.

```
%which dot[return]
/usr/local/bin/dot
```

In my case, everything is good. However, just like with `doxygen`, it is likely that your experience was not quite so satisfying.

```
%which dot[return]
dot: Command not found.
```

The `rehash` command probably will not solve this problem. The problem is that the shell (as set up by Apple) does not know that it should be searching in `/usr/local/bin/` for

commands. We need to add some directories to the variable `path`. At startup, tcsh will read a number of files (`/etc/csh.cshrc`, `/etc/csh.login` and `~/.tcshrc`, the ~ refers to your home directory). If you are interested in exactly when they are read, you should `man tcsh`. A good place to fiddle with your path is in `~/.tcshrc`. I am going to suggest you use `pico` to edit the file since this is a simple editing job. More complicated files should probably be handled with more complicated editors (like `emacs`, `vi`, `AlphaX` or `BBEdit`). Notice that `cd` is the same as `cd ~` (it will save you two characters of typing, which over your lifetime… well, probably still won't amount to too much).

```
%cd[return]
%pico .tcshrc[return]
```

The `pico` editor is very simple and even provides you with two lines of commands at the bottom of the screen. Here are some of the important lines that occur either in my `/etc/csh.login` or my `~/.tcshrc` files (watch the wrapping, there are three lines here, they start with `set`, `setenv`, `alias`).

```
set path = ( ${path} /usr/local/bin )
setenv DYLD_LIBRARY_PATH /usr/local/lib/graphviz
alias alpha "open -a /Applications/Alpha/AlphaX.app"
```

The `set` line will add the `/usr/local/bin/` directory at the end of the search path for files, once this is added, the tcsh should be able to find `dot`. In order to run, `dot` needs to determine where its dynamically loaded libraries are placed. The `setenv` sets up the information in an environment variable so that `dot` will be able to find this information while it is running. The next line show how to create aliases that can be used to open files in applications from the `Terminal`. In my case, the command `alpha` is equivalent to `open -a /Applications/Alpha/AlphaX.app`. To open `.tcshrc` in `AlphaX`, I can type the following command.

```
%alpha ~/.tcshrc
```

Unfortunately, the open command does not create a new file. If you want to create a file and then open it, one trick is to first `touch` the file first (`touch` updates the last modification date of the file and if the file does not already exist, it will create the file). You can use a semicolon to place two commands on a single line.

```
%touch blah; alpha blah
```

You need to add the first two lines to your `.tcshrc` file (the other line was provided because it may be useful). Once you type these lines, save the document by typing `[^O]` (control O, write Out) and hit `[return]` to accept the filename. Then exit `pico` by typing `[^X]` (control X, eXit). You can cause the commands in this file to be executed by using the source command.

```
%source .tcshrc
```

Now, when you ask `which dot`, the shell should find it (you changed the `path` variable so the shell automatically executed `rehash` for you).

```
%which dot
/usr/local/bin/dot
```

Reopen the Fish project and change the command in the Shell Script Build Phase to `doxygen  doxygen/Fish2.doxycfg` (notice the 2). The only difference between

`Fish.doxycfg` and `Fish2.doxycfg` is that `Fish2.doxycfg` has specified that we have dot (HAVE_DOT = YES) and that we will be using gif style images (DOT_IMAGE_FORMAT = gif). Now rebuild the project and then browse the new documentation.

CONCLUSION & BIBLIOGRAPHY

The steps above have focused on getting `doxygen` installed, but many of the bits and pieces are common techniques for installing programs on UNIX systems. Although I have not had a chance to work with `Xcode`, I suspect that the experience will be very similar to `Project Builder`. Hopefully you will take the time to try working with `doxygen` as well as finding other Mac OS X command line applications. If you find yourself using `doxygen`, please consider using the PayPal link from the `doxygen` website to show your appreciation. If you are really ambitious, `doxygen` can be extended to work with more languages than its current ones (it is open source). For example, an ambitious programmer could write code to support Objective C.

Here are the sources I used to put together this article as well as URLs mentioned in the article.

doxygen web site : www.doxygen.org/

GraphViz (dot) web site: www.phil.uu.nl/~js/graphviz/

introduction to tcsh: **Using csh & tcsh** by Paul DuBois (O'Reilly & Associates, Inc.)

my web site: www.cwu.edu/~montgoaa/

MacTech web site: www.mactech.com/