Defensive design

1. Introduction to defensive design

Application software often requires a user to input information into a system. This might include

- A username / password entry
- Contact details
- Current date
- Input into a search box
- Building up a shopping cart
- Entering data into a database

All these methods involve that most unpredictable component of any system - people!

Defensive design is the practice of anticipating every possible way that an end-user could misuse a system or device. During the defensive design process, methods are put in place to eliminate misuse. If this is not completely possible then it will aim to minimise the negative consequences.

You know that defensive design has not been good enough when

- The program crashes.
- The program behaves in an unintended fashion.

- Data security has been compromised.

This section will discuss 'defensive design' - the practice of anticipating problems and writing the appropriate code to deal with them properly.

Defensive design

2. User Interface considerations

The first port of call in defensive design is to decide on what kind of user Interface to code for. There is a fine balancing act with giving the user access to all the information and abilities that they might need, while at the same time imposing limits on the things that they commonly get wrong.

Menu Driven Interface

A menu-driven user Interface limits the user to being able to pick from a displayed list of choices.

The one above is a television / remote control Interface. The only actions that the user can perform is to use the up-down-left-right buttons on their remote control to navigate and a 'select' button to select an option.

A menu Interface can help to limit unexpected or invalid entries.

Another example are public booths with a touch sensitive screen.

Defensive design

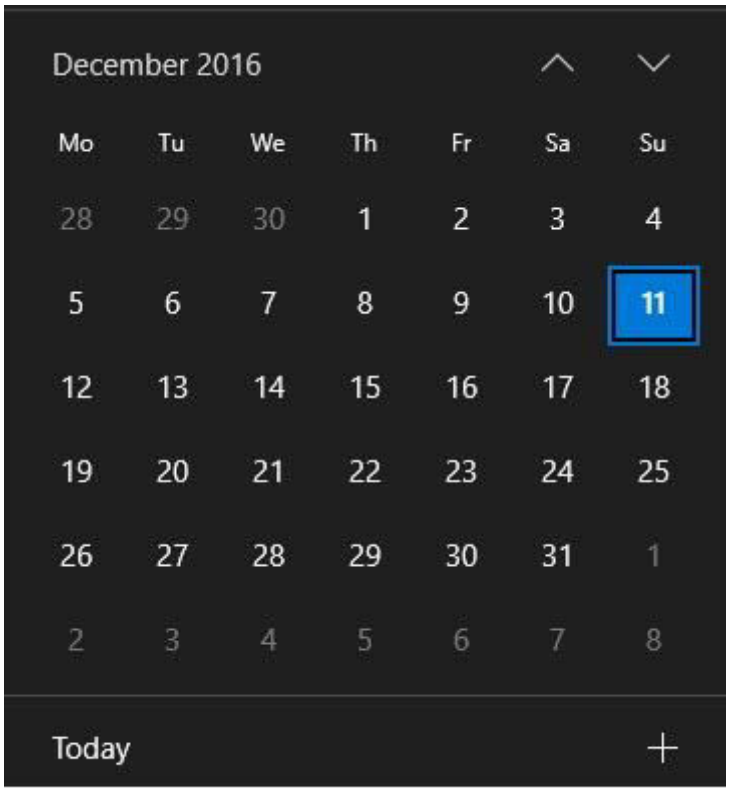
3. Screen Widgets

A hard-coded specific menu such as the airline selection screen you saw on the last page is difficult if not impossible for the user to get wrong but at times more flexibility is required. A good compromise is to accept information through graphical widgets, rather than input text.

A 'graphical widget' is a small self-contained object on-screen to allow data selection to be made.

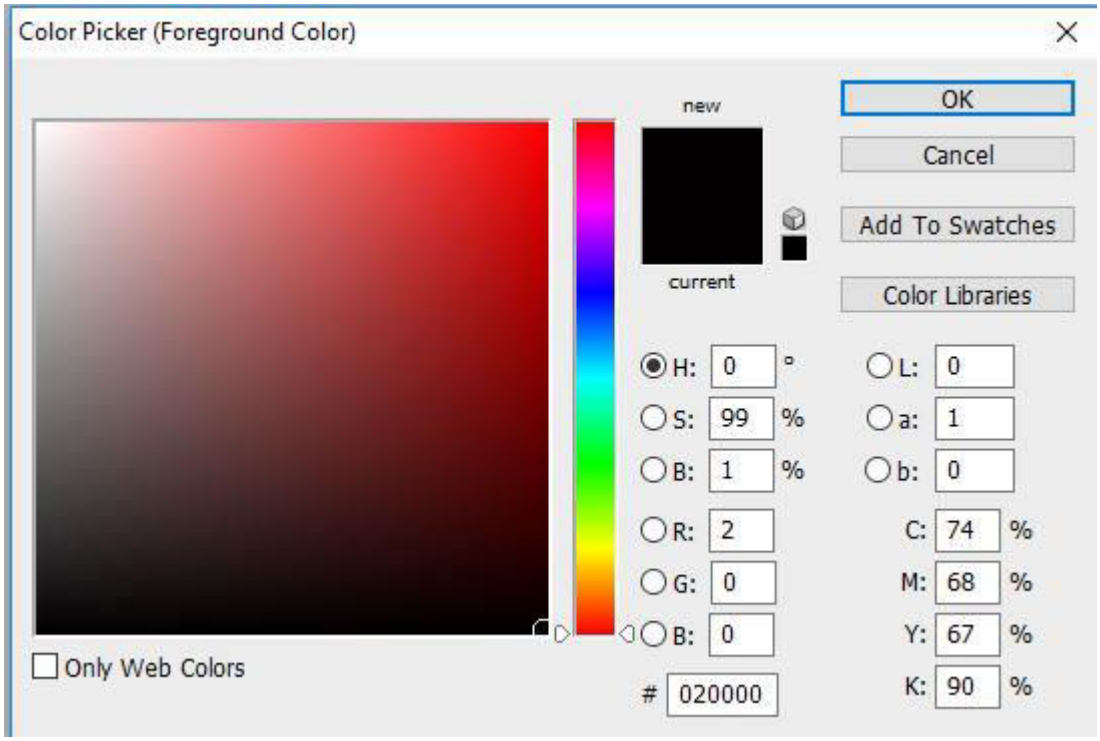
The calendar widget

This pops up a calendar from which the user can select a date. The advantage is that the date information can be encoded in any way convenient to the programmer, and the user cannot select an invalid value.



The colour picker widget

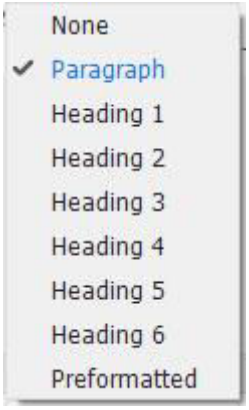
If the user needs to select a colour, then this allows them to pick the colour by eye or to enter a code if they know exactly what they require



Note that this widget has a number of input boxes - this means the user has the potential to get things wrong by entering an invalid value and so there must be checks in the code behind this Interface to deal with that properly. Input boxes will be discussed later.

Drop down list and spinners

The dropdown list presents the user with a number of items that they are allowed to select.



And a spinner allows a number to be selected by clicking on an up or down icon. Thus numeric data entry will always be valid.

Conclusion

We have only discussed three examples of widgets. There are many more. Their purpose is to limit user choice to only valid values.

Defensive design

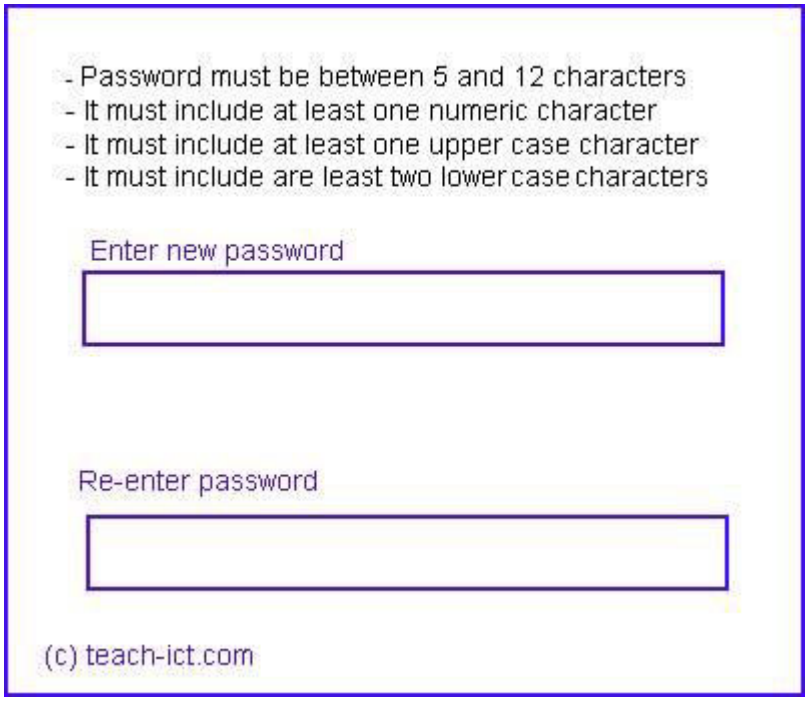
4. Good instructions and repeat entry

Text entry is the most flexible way of accepting inputs from the user, but also the most problematic. Users can mistype, or enter data into the wrong field, or simply input gibberish.

Defensive design involves making it as easy as possible for the user to get things right on the first try.

Probably the simplest thing for the programmer to do to reduce errors is to inform the user, with clear and unambiguous instructions on-screen, of what they need to enter.

Below is a password entry form. The user is presented with a set of rules that their password must follow- these rules should make it clear what kind of password will be accepted by the application.



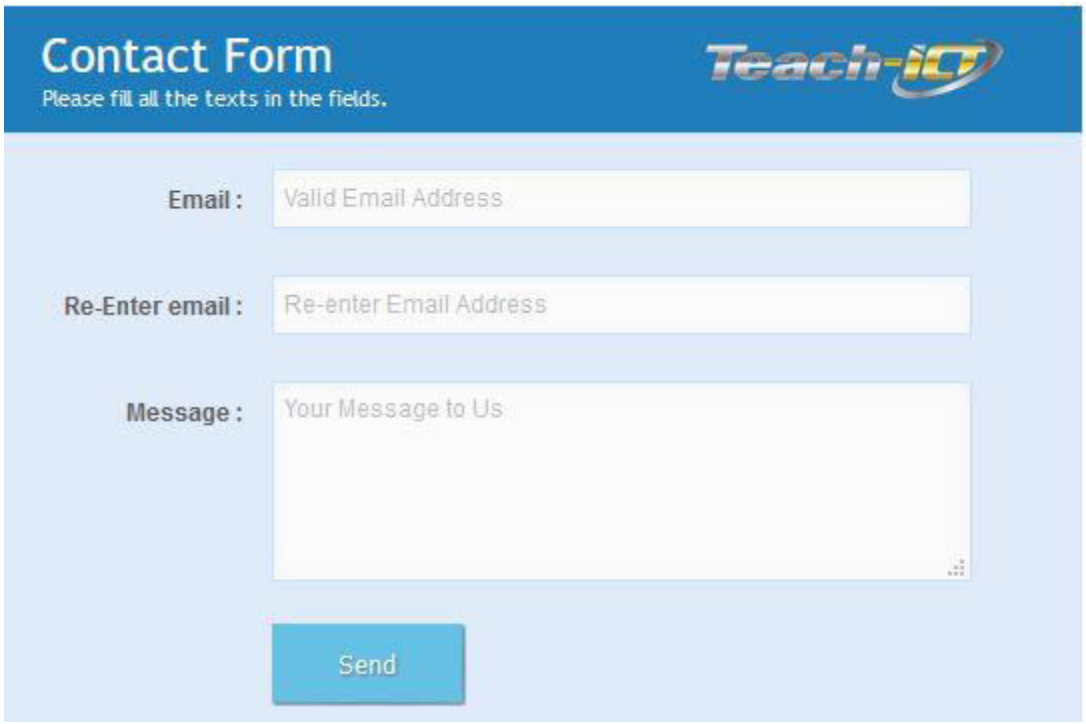
The user is also required to re-enter their new password so the program is able to check that it is what the user intended, as it is less likely that they make the same unintended mistake twice.

This technique is also useful for entering email addresses as they are so easy to misspell especially on tablet virtual keyboards with predictive text switched on.

Defensive design

5. The input box

The data input type that is most vulnerable to problems - deliberate or otherwise - is the input box and access to keyboard entry. For example the picture of a form below includes three input boxes for data entry.



As the user is now free to try and enter whatever they like. The program handling the input needs to include some defensive measures. These defensive features are called '**validation**'.

Validation seeks to ensure that the data entered is valid for further processing by the program handling it.

The four validation methods are

- Checking and limiting the **Length** of the data
- Checking the **Range** of the data
- Checking the **Type** of the data
- Checking the **Format** of the data

These validation methods are discussed on the next couple of pages.

Defensive design

6. Validation: Length and Range

Length validation is checking to see if the entered value is within the allowed number of characters. For example, password forms may reject an entry if it is less than three characters, as it would be too short for security. Empty fields (those with a length of zero) may also be rejected.

Below is a well-designed text input box for the user to enter their comments. Note it has three things to help the user. It has a default instruction inside the

box, which disappears as soon as an entry is made. It has a warning that no HTML code will be accepted, pure text only, and it has an active character count counter that tells the user that only 400 characters will be accepted..

If the validation rule is broken, the programmer has two options

- Inform the user of the problem and allow them to re-enter the data
- Modify the input to fit the rules. For example - cut off (truncate) the excess characters if it is too long, or pad it out if it is too short.

The truncation option is useful for message input boxes such as the one above. For example the user may have typed (or copy-pasted) a long message, but the program truncates it to say 400 characters.

A text prompt is useful to explain to the user what is needed within each input box.

Range validation

Range validation is used when inputs have to fall between certain values. If a form asks for the user's age, then it makes sense to check that they do not input a negative number, or a decimal. Defensive design in this case is writing the software code to check that these conditions are met.

Defensive design

7. Validation: Type and format

Type validation

Type validation is checking that the input conforms to the allowed data types.

For example, perhaps the input can only be a numeric value and so alphabetic or symbol characters are rejected. The software must be coded to deal with these inputs without causing a program error.

Format validation

The data format is often important, inputting data records into a database often needs to check that the input is in the correct format.

For example an UK National Insurance number has the format XX999999X where the first two and the last characters must be letters whilst the middle characters are numeric.

In this instance, the code will examine the input and check that it fits the pattern correctly.

Defensive design

8. Validation example code

As an example of defensive coding, it may be useful to see what real validation code looks like.

Consider that the user has visited a web site and filled-in an online form. As soon as the 'submit form' button is pressed the data is sent to the server to be processed.

The picture below shows a fragment of form validation code written in a computer language called PHP which is very popular for processing data on web servers.

There is no need to understand the code, but to highlight a few things :-

Lines 105 and 106 are going to handle the minimum length of each input box on the form.

Lines 110 and 111 are going to handle the maximum length case.

Lines 115 and 116 are going to handle required fields in the form.

Lines 120 and 121 are going to handle invalid duplication.

The complete form validation function has a thousand lines of code in it which shows how much effort has to go into defensive design in real situations.

```
92      ##-------------------------------------------------------
93      ##                          Main Form Validation Caller
94      ##-------------------------------------------------------
95      function validate( $input = '' )
96      {
97          if( is_array( $input ) )
98          {
99              foreach( $input as $key => $row )
100             {
101     // echo "\$input[$key] => $row[3].\n";
102                 $action = strtoupper( $row[0] );
103                 switch( $action )
104                 {
105                     case 'LENMIN' :
106                     case 'LENGTHMIN' :
107                         // ( action, min, $name, min error me
108                         form_handle::valLenMin($row[2], $this
109                     break;
110                     case 'LENMAX' :
111                     case 'LENGTHMAX' :
112                         // ( action, max, $name, min error me
113                         form_handle::valLenMax($row[2], $this
114                     break;
115                     case 'REQ' :
116                     case 'REQUIRED' :
117                         // ( reference, value, name, error me
118                         form_handle::valRequire($row[1], $thi
119                     break;
120                     case 'DUP1' :
121                     case 'DUPLICATE_BAD' :
122                         // ( reference, query, direction, err
123                         form_handle::valFind($row[1], $row[2]
124                     break;
125                     case 'DUP2' :
```

Defensive design

9. Deny list and Allow list

Allow list

An allow list is a list of data that the application will accept as valid.

Deny list

A deny list is a list of data that the application will reject.

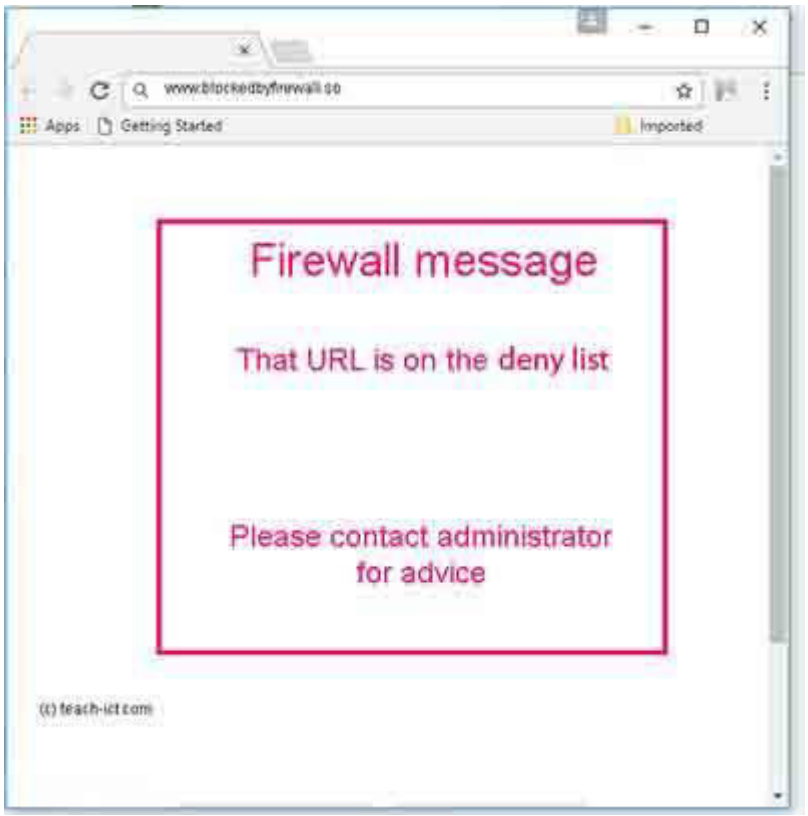
Uses

Allow lists and deny lists are often used with firewall applications and email filtering applications.

A firewall is an application that helps protect a network from intrusion or to limit what network users can access online.

One of the things a firewall checks is the URL sent from a web browser within the network, such as the one shown below.

When the user types in the URL, effectively this is an *input* to the firewall application even though the user may not be aware of it - as far as they are concerned they are just using a browser. Should the firewall reject the URL because it is on the deny list, then it returns a custom web page back to the user to let them know that it has been rejected.



The allow list may contain a list of URLs (web addresses) that the firewall will let through, which means the network user will be able to access that site. The deny list may contain a list of banned URLs which will be blocked.

It is easier to create an allow list than a deny list because it is much easier to define what IS acceptable than to try and anticipate what is NOT acceptable.

Defensive design

10. Authentication

Authentication is a way of confirming that the user is authorised to access the system.

Username & Password

The most common way of doing this is to ask for a username and password. The entered details are then checked against a database containing valid accounts.

Two-factor authentication

For high security sites such as financial services 'two-factor' authentication is becoming popular. This means that after the user enters a valid user name and password, the system sends an SMS text 'authentication code' to the designated mobile phone. They then have to enter this as well.

The idea is that an unauthorised person is less likely to have the mobile phone to hand as well.

Biometric authentication

This method checks some physical feature of the authorised person such as their fingerprint. The user puts their thumb on a fingerprint entry device, the data is sent off to a database and checked against their valid data.

Defensive design

11. Summary

- Defensive design seeks to minimise the occurrence and effect of user errors
- A menu Interface can minimise possible user errors
- A menu Interface may not be flexible enough for the application
- A 'graphical widget' allows data selection to be made with less chance of user error
- Including clear instructions on-screen helps the user to input valid data
- Inviting an user to repeat the entered field helps reduce typing errors
- Validation is a technique to ensure that data input is valid
- Validation includes checking length and range of data
- Validation includes data type checking and data format checking
- Defensive coding can take significant effort, but is necessary for most applications
- Allow list defines a list of acceptable data
- Deny list defines a list of data that will be rejected
- Including encryption within the application helps with data security

Maintainable code

1. Introduction to maintainability

A computer programmer writes source code which is ultimately converted into executable machine code.

Once the machine code has been created and stored in a file, you could in theory just throw away the source code. After all, the program will run perfectly well without it.

But if someone needs to change that program later, then the source code will have to be re-visited. So it is usually a bad idea to discard the original source code.

A program may need modifying for any of the following reasons:

- To add a new feature
- To fix a bug
- To improve the speed or performance
- To port it so that it will run on a different CPU or Operating System

The process of ensuring that code can be easily amended or updated is called **Maintainablility.**

There are several standard techniques to help with maintainability that every programmer should follow, regardless of which programming language they use to write their code.

We will discuss these techniques in the following few pages, using a worked example to demonstrate.

Maintainable code

2. Comments

Programming languages generally allow you to mark blocks of text within a program which the computer will ignore when it runs or compiles the code. These are called 'comments'. They can explain what a particular line or block of code is doing.

By using comments, people who look at your code will be able to work out what it is supposed to do. Even things that seem obvious while you're writing them can be difficult to follow in a few weeks or months. So it is good practice to include comments after each of the following:

- **Variables and constants** - While you should use descriptive names for every variable and constant, comments can provide an extra hint about what you intend to use them for.
- **Procedures and subroutines** - These, more than any other type of code, cause the program flow to jump around between sections, so you should include a description up front about what data each subroutine is calling, what it is doing to that data, and what it returns to the main program.
- **Throughout the program** - Good programmers include enough comments that the reader could, if they wanted to, rewrite the program following just the descriptions provided. At the same time, comments should describe what a block of code is doing *overall*, rather than specific descriptions of what every line is for.

Maintainable code

3. Headers

One very popular use for comments is to add a header at the very start of the code. This header can describe the program as a whole.

Good information to have in a header includes:

- The **original author** of the code. This way if the reader has any questions about the program, they know who to go to for answers.
- The **date** that the program was written. This gives the reader some idea of where to start with optimising the code. Features and tools that were developed after the program was written will probably not be used, after all.
- The **version number** of the program. Useful programs are often updated multiple times, with each new "version" making different changes. By comparing the version number in the header to a documented "version history" of the program, the reader can be sure which version of the program they are looking at and which bugs and features have already been amended.

Maintainable code

4. Formatting

Even the clearest writing can be difficult to follow if it is formatted badly. Imagine trying to read a book if there were no spaces between the words and no paragraph breaks.

Whitespace

Similarly, when writing code it is important to break up blocks of code visually.

Compilers for most languages ignore whitespace entirely. And those that do not, like Python, require the programmer to follow good practice. Whitespace and indentation are powerful tools in helping to make code easy to follow.

Blocks of code should be separated by empty lines, with each block consisting of related lines of code that work together to perform a specific task.

For example, a block of code that sets the appearance of a table might have

- one line setting the width,
- one line setting the height,
- one line setting the font of the text inside

All of these are related to the same task. When the code moves on to describing how to do something else, it's time to start a new block

Maintainable code

5. Indentation

We covered good practice for indentation back in the module on algorithms, and the same methods should be applied to actual code. Nested conditional statements and loops, for example, should be indented to make it clear where one starts and another ends, like this:

```
START
IF x < 10 THEN
  y = 1
  WHILE y < 20
     PRINT (y)
     y = y + 1
  END WHILE
       ELSE:
           PRINT ("Nothing to see")
END IF
END
```

As you can see, indenting makes it much clearer which lines are part of the WHILE loop, which lines are part of the if statement, and which are neither.

Next, we will go through an actual example of code and show you how all of these tips on maintainability can make things much easier to follow.

Maintainable code

Maintainability: Worked example

Introduction

Rather than just stating the techniques, we thought it would be better if you followed a worked example of real code.
Let's imagine that you have just joined a new company as a programmer and the first thing they do is ask you to alter some source code that a previous employee has written. The computer language used is PHP but that does not matter, as the same approach should be used whether you are writing in Python, Java, Assembly language or anything else.
We begin with source code that has been deliberately made to be as horrible as possible in terms of maintainability.

Look at this code:

The link below will show the code in a new tab in your browser, we recommend you drag that tab out of the browser to create a new view so you can see it side by side with this page.

Please open this link IN A NEW WINDOW to view the code

Question: With just a brief view of the code, can you tell what this code is supposed to do?

Answer: Very unlikely, even if you were expert in PHP. The programmer has given no explanations or structure, making it very difficult to follow. But it is perfectly working code! Which means you can write perfectly working code but it takes skill and professionalism to make it maintainable. We shall build up this code to show how this is done.

As an aside, if you write programs just for yourself, even then, you should follow good programming practice. You may re-visit your own code some years later by which time you no longer remember why you wrote it that way. So use professional coding practice for all the programs you write.

---

1. Header

Solution: The original programmer should have included a header comment stating its purpose.

Look at this code:

Please open this link IN A NEW WINDOW to view the code

Can you spot the header? It's in line 2.

Now the code tells the reader the bare minimum of information - that this program is an RSS reader. RSS is a web technology to send news to its subscribers, so this code is intended to read a RSS newsfeed.

But it's still rather bare.

Look at this code:

Please open this link IN A NEW WINDOW to view the code

From line 3 and on, the header has now been expanded to include several new details. It now mentions the **original author** of the code, the **date** that the program was written and the **version number**.

This is much more useful.

---

2. Formatting

As we discussed on earlier pages, one of the key tools to making source code readable is formatting and whitespace. The human eye is drawn to patterns, and blocks of text broken by empty lines intuitively indicate that different things are being discussed in each block.

The php code we've been looking at so far is completely unformatted. Let's see what it looks like with just some simple line breaks:

Look at this code:
Please open this link IN A NEW WINDOW to view the code

In this version, the programmer has added blank lines (lines 10, 15, 21, 38) to provide some visual space between code blocks.

Now it is apparent where the FOR, IF, and FUNCTION blocks begin and end. This improves readability dramatically, but it could be improved even further.

Functions (subroutines \ procedures) are self-contained blocks of code that are called from other parts of the program. It is very useful to easily see where it starts and ends, because you can see if the function syntax is correct (in PHP it needs to start and end with {}). The return value - if any - is also easily visible.

Look at this code:
Please open this link IN A NEW WINDOW to view the code

As you can see, everything inside the function is now indented to illustrate the relationship of those lines of code.

The opening { bracket on line 11 clearly has the correct } closing bracket on line 40 according to the syntax rules of php. The return item on line 39 is also obvious. All this from a single indent of the code.

There's still more to indent, though!

Look at this code:
Please open this link IN A NEW WINDOW to view the code

Now the conditional statements and loops (IF and FOR respectively) within the function are also indented to indicate where they start and end.

3. Comments and labels

Now that we've formatted the code and separated all of the blocks of code by their purpose, it's time to make it clearer what those purposes *are*. This is where commenting comes in handy. To mark a section of code as a comment in this language, you use the characters // (it is different in other computer languages but they all have a way to comment code)

Anything after // on that line will be ignored by the computer when it comes time to run or compile the program.

The first thing to comment on is the purpose of the functions. A program often has many functions - dozens is not unusual. The programmer should explain what each function is intended for. The description should be in clear language and be fairly non-technical if possible.

Look at this code:
Please open this link IN A NEW WINDOW to view the code

In this case line 11 to 15 explains the function and how it is to be used.

Next up, changing the names of the functions and variables to be more descriptive. The programmer might have known what the function "f" meant (lines 8,16), and what the variables "a" (lines 22,27) and "c" (lines 9,17,19) were, but we certainly don't!

Using descriptive names makes it much easier to locate where else in the code that function or variable is mentioned. And also what is the purpose of the item.

Look at this code:
Please open this link IN A NEW WINDOW to view the code

As you can see, function "f" (line 16) is now named "getRSSdetails", since that's what it's doing - it is getting RSS details from a database. All references to the function in the source code have also been updated (line 8)

The variable "a" (line 22) is now "num_results", since it's showing the number of results and this is used in the for loop as well on line 27.

And the variable "c" is now "db" - short for "database". So the instruction on line 9

$db->disconnect

should be pretty obvious that a database is being disconnected.

You could be even more descriptive than this with your naming scheme, but at some point it becomes tedious to type out long names.

The last stage is to add general comments throughout the code, just to make things easier to follow.

Look at this code:

Please open this link IN A NEW WINDOW to view the code

Now it is time to add even more comments. But intelligent ones!

If you were a professional programmer, imagine that your source code was up for review on a big projector screen (not unusual in a real company doing a code review!) with lots of your fellow programmers in the audience - what would you say about your code as you walk through it?

This is what your comments should contain - not minor technical detail such as 'this increments a variable', but your thinking behind an item or block of code if it is significant.

As an example, line 21 says

```
if (msql_affected_rows() > 0 ) { // deal with no records being returned
```

The comment 'deal with no records being returned' has nothing to do with the actual line of code, it is assumed that the reader is familiar with php and knows what the code instruction itself means, but the comment is saying what the programmer was intending this code to do. This really helps with spotting logical errors if that bit of code does *not* do what the comment is saying it should.

If you were unfamiliar with the programming language - php or anything else - look at the comments. It is as if the programmer (John Brown in the header) is guiding you in his thinking as the code was written. Good comments should not matter about the actual code being used.

---

4. Comparison

Now that we're done making this code more maintainable, it's worth comparing it to what we started out with. The two styles of coding are shown side by side. They do exactly the same thing, the computer does not care about comments, indenting, formatting - but you or your fellow programmers will!

Please CLICK HERE to view the code

Hopefully, you will appreciate that the code on the right is going to be far easier to maintain than the obscure code on the left.

For example, if the code had to be re-written in Java rather than PHP, it would not be too hard a job if good coding practice has been used.

Maintainable code

7. Conclusions

Most software tends to need adjustment even after the first version has been launched.

To make this task easier, the maintainability of the code needs to be of a high standard.

In order to do this, then good coding practice should be followed. Including

- Extensive comments to explain each part of the code and its overall purpose
- Effective use of indentation to easily see related blocks of code
- Effective use of blank lines to visually split up the source code into related parts
- Using sensible and meaningful names to variables, constants and functions

The reasons include

- To make it understandable i.e. to allow someone to look at the code and easily understand its purpose and function
- Software bugs are more easily identified and fixed
- To allow version control i.e. a record of changes made between iterations of the program
- To allow the code to be amended or expanded

- To increase portability between programming languages and operating systems