How to Write Good Software

Introduction

There are many good pieces of software which exist out there but for every good piece of software there are probably at least twice as many bad pieces of software. In this article I will try to help you to understand the differences between good and bad software and present you with some general advice for designing and implementing quality software.

What is Good Software?

First let me begin by definding exactly what I mean by good quality software. Good software should be functional, usable, reliable, robust and efficient. A good implementation should be portable, maintainable and reusable. So what do each of these terms mean? Let me go through them one by one and describe to you what they mean.

Functional software should meet all of the functional requirements. This applies to software engineering where a client defines what exactly the software should do. It should do these things but no more. Adding extra functionality which is not in the functional requirements could impair reliability, efficiency or maintainability. It is worth noting however that sometimes adding greater generality can cost nothing at all.

Usable (or user friendly) software should be exactly that. It should be easy to learn and easy to use, helping the user to work faster and avoid making mistakes while providing as enjoyable experience as possible.

Reliablity can be measured. Perfectly reliable software never fails, and this is what we should always strive for, however this is hard to obtain. Highly reliable software fails infrequently and non catastrophically. A failure is any behaviour which violates the specification such as loss of data, wrong output, abnormal termination and so on. A catastrophic failure is a failure which could cause loss of life, damage to health or the environment or the economy. Mission critical software is software whose failure could be catastrophic. It must be highly reliable.

Robust software should not fail and should behave reasonably even when unexpected situations such as badly formed input, shortage of memory and so on occur.

Efficient software meets all space and time requirements and uses both space and time economically. Sometimes worthy tradeoffs are made such as using up more space for increase in speed (memory is much cheaper than processor cycles). However efficiency should not be sacrificed for important attributes such as reliability or robustness.

Maintainable software should be well documented, readable, modular and well. This makes it easier for other developers to be able to understand how the software works, and to be able to easily make changes in the software without having to completely redesign it.

Portable software will run on a variety of different hardware architectures and on different operating systems without little or any change to the code.

Reusable software components are general enough to be included and used in other pieces of software without any change.

Top Down Design

Top down design is the process of breaking down a program, given the functional requirements, in to individual software components called modules. Considering each module in turn, if the module is simple enough then it can be directly implemented as a **worker module**. If it is still complicated and requires further decomposition then it would be turned into a **coordinate module** which coordinates simpler worker modules. Typical types of modules are procedures, functions, classes, module files and so on. Note that one of the weaknesses of top down design is that it often leads to widespread data type coupling or common variable coupling (explained shortly).

- You should aim to keep procedures and functions below 60 lines of code so that it can be viewed on one screen without having to scroll. This gives anybody looking at the code a better picture of the structure and the code thus allowing them to understand it more easily. A procedure or function with more than this amount of code would probably be doing too much anyway. This is not a set in concrete rule which must always be followed. There are some cases where it is necessary to write long procedures which may span several screens. For example, a regular expression parser may contain lengthy procedures.

- Try to keep the amount of code inside classes, packages, module files or other similar modules to no more than a few hundred lines of code (say, about 600). If the code is any longer than this, then this suggests that the module is doing more than it should and should be further decomposed. Again, this rule isn't set in concrete and there are cases where it would be perfectly acceptable to write a several thousand line module consisting of hundreds of procedures. This is just meant as a general piece of advice.

- The above advice helps you to keep your modules **cohesive**. Cohesive modules do just one thing, and do it well. A single function should not do X and Y then Z but should just do one simply specified thing. If a module does more than one thing, consider further decomposition. This is especially true if it does many unrelated things.

- Keep modules as independent from each other as possible so that changes in one module impact as little other code as possible. Modules which are insensitive to changes in one another are called **loosely coupled**. Having the modules loosely coupled (as opposed to **tightly coupled**) to each other makes the program more maintainable and easier to understand.

- Think about the software as having many different levels. Modules on the bottom (worker modules) should take care of detailed processing. Intermediate coordinate modules should rely on the worker modules to do the work for them, but should coordinate control and data flow amongst worker modules. The main program relies only on coordinate modules.

Having a good design where modules are cohesive and loosely coupled will help you in many ways. When implementing, modules can be coded, tested, debugged and documented independantly of each other. One programming team could, given a specification, work on module X while another programming team, given a specification, could work on module Y. At the same time, a third programming team, using both specifications of modules X and Y could write the coordinate modules which coordinate them. Each of the three programming teams can work seperately. They can then easily write test cases and thoroughly test and debug their module before integrating it into the system as a whole. It also helps maintainability a lot. If the software needs changing, you can simply locate the module which need changing then change, retest and reintegrate these modules into the system. The loose coupling means that you won't have to go through the rest of the code changing bits all over the place.

More on Cohesion and Coupling

A module is cohesive if it has a simple specific function and every part of the module is essential to that function. A module is uncohesive if it has two or more distinct functions (and even moreso if they are unrelated to each other) or it contains operations which could be reasonaly delegated. All of the elements of a cohesive module should be tightly coupled to it. If they are not, then the module is probably uncohesive. You can sometimes decide how cohesive a module is by trying to describe it as simply as possible. If you can say "Module X does Y" then it is likely that module Y is cohesive. If however you end up saying "Module X does Y and Z" or "Module X does Y then Z" or "Module X does Y or Z" then module X is not cohesive and should be decomposed in to 2 seperate modules; one which does Y and the other Z. Moderately cohesive modules are sometimes acceptable, where the module does 2 distinct but related tasks.

Coupling is a measure of how much modules depend on each other. If a change to module X forces large changes in module Y then module Y is said to be **tightly coupled** to module X. If a change to module X requires very little changes in module Y then module Y is said to be **loosely coupled** to module X. If a change in module X requires no change to be made in module Y then Y is **uncoupled** to X. If modules are tightly coupled then changes in 1 module can easily force changes in other modules, and changes in those modules may require changes in other modules and a chain reaction occurs. This makes for very poorly maintainable software where even small changes to the design can lead to massive overhauls of the code. This can be avoided by making modules more loosely coupled. Furthermore, loosely coupled modules are more reusable as they can be easily added to other programs without requiring the addition of many other modules which they are tightly coupled to.

There is always some amount of unavoidable coupling however. If module X calls module Y, passing parameters and such, then X is **normally coupled** to Y. If X only relies on the specifications of the functions it calls in Y being the same, then it said to have **loose normal coupling**. If Y makes some assumptions about the data passed in to it or if X makes some assumptions about the data Y passes out of it then the modules have **tight normal coupling**. Loose normal coupling is usual and necessary. If you suddenly change a function name, or change the parameters a function expects then anything calling that function is obivously going to be affected. However, notice what is said about tight normal coupling. If you write a function and make assumptions about the parameters passed in to it, then you are making it tightly coupled to anything which uses it. You should validate parameters and so on to avoid this. Similarly, if you make assumptions when calling a function then your code is now tightly coupled to that function. If somebody were to change that function, your code may require changes in order for it to continue to work.

Constant Coupling occurs if a module uses a global constant. If the module contains the literal value of this constan or if the module also makes any assumptions about the value of the constant (for example: the constant will always be less than 10) then it is tightly coupled to the constant. If the module refers only to the constants name and makes no assumptions about which value it contains then the module is loosely coupled to the constant. To avoid tight constant coupling you should always declare constants in just one place, then refer to them only by name in your code. This makes the code more maintainable because if the value of the constant must be changed then only the declaration of the constant needs to be changed and the program should still work as expected.

Data Type Coupling occurs between a module and a data type if the module uses that data type. If the module depends on the representation of the data type then it is tightly coupled to the data type. If however the module only uses the name of the data type (for example, declare variables of this type, pass it to functions and such) then it is loosely coupled to the data type. Try to avoid declaring data types more than once and instead declare them just in once place.

Common Variable Coupling occurs when several modules access one common variable. Common variable coupling is always very tight because the modules using it depend on both the data type of the variable and on each other to initialise, update and inspect the variable in an expected way. Try to localise any common variable coupling within modules such as classes or packages (or whatever) by giving them a scope private to that class/package/whatever only. Try to avoid global variables.

Writing Efficient Software

There are a few easy steps you should always follow to avoid inefficiency. I will simply list these, and give some small examples first. Note that following these steps does not necessarily lead to efficient software however. Correct choice of data structures and algorithms and other factors also come in to play soon. However, as a good start, try to:

- Avoid unecessary computation. Don't bother computing results which are never used.
- Avoid repeating the same computation. The common cause of repeated computation is loops so make sure that you aren't doing the exact same computation inside loops especially.
- Don't create unecessarily large data structures.
- Don't declare unecessary variables.
- Keep variables in scope only when needed. Don't declare a global variable, which will be in memory for the duration of the program, where a local variable will do.
- Deallocate any allocated memory (heap variables) as soon as it is no longer used and allocate it only when it is needed.

Example:

```
if foo(a) then
 ...
elseif not foo(a) then ' Unecessary repeated calculation. Just use "else".
 ...
end if

for (i = FOO; i < BAR*BAZ + 10 - 'A'; i++) { /* calculations performed over and over */
 ...
}
```

Note: Sometimes compilers will fix your mistakes for you. For example, any good C compiler should move the calculation in the above for loop outside of it in the compiled program.

Choice of programming language can definetly play a large part in writing efficient software. For example, assembly code has a relative running time of ~1, C of ~2, Java of ~4, VB of ~5 and so on. This means it's possible to write programs in assembly which could be potentially much much faster than if we were to write them in, for example, VB. Usually the faster the language, the less easy it is to make maintainable and the harder it is and the more time it takes to implement, however. Sometimes the extra effort required to write in a lower level langauge would be better spent fine tuning the speed critical parts of the program written in a higher level langauge whith more maintainable code.

There is a well known saying that about 90% of the running time of a program is spent executing just 10% of the code. Think about this for a second. If we make the most time consuming 10% of the code run twice as fast then we cut the total running time by about 45%. If we spend 10 times as long making the entire program run twice as fast then we cut the running time by 50%. **Tuning** is the process of optimising only the most time critical pieces of code. This is only possible if we know how long each piece of code takes to execute. There are tools which give us this information called **code profilers**. Alternatively, we can manually insert timing code in to the program.

As I mentined above, the choice of algorithms can affect the speed of the code in huge ways. A bubble sort may take literally several years to sort massive data sets where a quicksort may take only a few milliseconds. On the other hand, if the data sets are only going to be very small then perhaps the extra effort and code complexity required to write a quicksort may be unecessary. You can decide how good an algorithm is by examining how many times operations are performed. If an algorithm takes a constant time regardless of the size of the data set then it has O(1) (read 'order one') complexity. For example, multiplying 2 numbers takes the same time no matter what the 2 numbers are. Multiplication has O(1) complexity. An algorithm which halves the amount of work it must do on each iteration has O(log n) complexity. Such an algorithm would be a binary search. The time taken for this algorithm increases very slowly as the size of the data set increases. Increasing the data set by about 1000 times is only going to take about 10 times as long. Algorithms with O(n) complexity take linearly increasingly amounts of time as the data set increases linearly. So for example a linear search (which has O(n) complexity) on 1000 items will take 100 times as long as a linear search on 10 items. Try to choose an algorithm appropriate for the size of the data sets you are expecting. Don't pick a bubble sort for sorting 10 million items.

As also mentioned above, the choice of data structure can also have significant effects on speed. If you were to store data as a binary search tree then performing a search is very quick, with only a small penalty for inserting and removing items. A hash table allows elements from large data sets to be accessed very quickly. Red-Black binary trees could be a better choice over a binary search tree when the data set is likely to result in an unbalanced binary search tree.

In general, a combination of good data structures, good algorithms, following the simple rules given above, and optimising the most speed critical 10% of the code will result in very efficient code.

Writing Robust Software

Robust software should be able to handle ill formed input, unexpected situations like running out of memory, unexpected values and so on. Functions and procedures should validate values passed to parameters and code which expects specifically formatted input from files or from the keyboard should validate that input too. This is **defensive programming**. Modules should respond to exceptional situations either by handling it silently in a reasonable way, returning some kind of status code to the client/caller or by raising an exception/error. Status flags require no special language features however status flags are often not checked or ignored and when they are checked, the code usually has exception handling code entangled in amongst normal code making the program harder to follow. Exceptions require support for exceptions in the language but they allow the exception handling code to be seperated from the normal program code. Exceptions usually carry on up the call stack until either they are handled, or they reach the top of the call stack in which case the exception is displayed to the user in some form, or the program ends. Exceptions should be handled before they reach the top of the call stack. Batch programs should log errors to, for example, a file. Interactive programs should display errors immediately using informative error messages. For example, if ill formed input is detected, inform the user and re-prompt for the input.

Writing Reliable Software

Software components should all be tested thoroughly. To test a software component, we run it with a selection of test cases for which we know what the result should be, and compare the results with what we know the result should be. The aim of testing is to expose faults so a successful test is a test which tries to exposes some fault in the software. A test case which would fail to expose any kind of fault is a waste of time and energy. It is very important that the output should be known before hand, or predicted from the specification but **not** from the implementation. Keep the test cases for **regression testing**. If you decide to change a component later, or if during the maintenance stage a component must be changed, then the same test cases can be run on it after the changes have been made.

Functional testing treats the component as a black box, formulating test cases from just the specification. This is analogous to test driving a car. Systematic testing tests each part of the code in a systematic way. This is analogous to a mechanic performing diagnostic tests on the car. Functional testing could be performed by the user whereas systematic testing must be done bya programmer who knows how the code works (refer again to the car testing analogy with the user and the mechanic). When choosing test cases, choose normal inputs to test that the program behaves expectedly in normal circumstances. Also choose boundary conditions to test whether or not the program behaves correctly for extreme values and so on. If modules can raise exceptions, return status flags indicating failure of some sort, or handle exceptional cases silently then test these. When doing systematic testing, ensure that every statement can be executed sometimes. Also make sure that every condition can either true or false sometimes and also that every subcondition can either be true or false. Finally, make sure that loops always terminate and that loops can either be executed 0, 1 or more times.

Testing only aims to prove the incorrectness of code. It can not prove the correctness of code. Only exhaustive testing can prove the absence of faults but this is generally impractical. Thorough testing will however increase confidence that the code is correct. A correctness proof is a formal proof that shows a given implementation meets a given specification. There is no absolute proof of correctness since the proof itself could be flawed, or the spefication wrong. For small modules that can be formally specified (such as a sorting algorithm), correctness proofs are feasible however for large systems they are not. Correctness proofs can only be performed by somebody with strong advanced mathematical and programming skills.

Writing Portable, Reusable Software

Portable software is software which is loosely coupled to the platform (machine and operating system) on which it was designed to run. Causes of unportable software are writing in machine speficif languages such as assembly or machine code, using non-standard language dialects like J++ or Turbo Pascal, dependancy on the hardware (peripherals, specific instruction sets) or dependancy on data representation (especially characters and numbers). You can avoid writing unportable code by choosing standard high level languages such as java, or by using lower level languages such as C or C++ with care. If platform independencies can not be avoided then try to localise them as much as possible so that only small parts of the code must be changed to port the software to a different platform.

Reusable software components must be cohesive, loosely coupled, portable and suitable for use in a wide variety of programs. Languages which support things like generics allow us to write, for example, sorting modules, linked list modules, binary search tree modules, hash table modules and so on which can be easily inserted in to other applications and reused. Some components, if they don't use langauge dependant features like exceptions or generics could be turned in to standard libraries (dlls on windows, object files etc) which can be easily reused in a wide range of different programming languages.

Writing Maintainable Software

To write mainaintable software you definetly must have a good coding style. Things such as layout, naming, commenting and consistency are important aspects of coding style. The code layout should reflect its logical structure using indentation and to some extent, the ordering of functions and procedures inside code files. Seperate them by one or more blank lines so that it it easy to distinguish where one ends and another begins.

Comment where necessary, but remember. Too much commenting can be just as bad as too little. Add comments to the top of code files detailing when the file was written or last modified, who the author(s) are, the license (if any), and what the purpose of the file is. Add comments to the top of procedures and functions detailing what it is supposed to do, whether or not it raises any exceptions and what the parameters and return values mean. Do **not** describe how it is implemented in this header. Use comments to describe the implementation inside the body of the code but be careful not to state the obvious. Explain how things are supposed to work instead of just repeating what the code does. The comment should reveal the intention of the code.

Use appropriate naming conventions. Pick a convention and stick to it so that your code is consistent. Use descriptive names where possible. For example the function “IsUpperCase” is much more descriptive than “uc”. Notice how the name of the function alone implies that it returns a boolean, because of the word “Is” at the beginning. This also allows it to be inserted naturally inside conditions where it can be read almost as natural language. “if (IsUpperCase (MyString))”.

Document your code. The use of structured diagrams or UML diagrams can be greatly benneficial to anybody trying to understand your code. Provide documents for interfaces of classes and so on (which methods it exports and how they should be used). A specification of a software component consists of an outline of the interface. For example a function specification should detail the name of the function, the types of the parameters it expects and the type it returns. A class specification may contain specifications of functions and other methods exported. A specification also provides a description of what th software component does and how components relate to one another. A good specification must allow other programers to use the component without having to look at or know the implementation. It should also allow other programers to be able to implement the component themselves without knowing how it is to be used. A good specification should be concise and precise.

Conclusion

There are no hidden secrets or black magic involved in writing good software. All you need to do is write loosely coupled, cohesive modules with clean, well commented code. Choose appropriate data structures and algorithms. Spend time testing and documenting your code.