Defensive Programming

Nikolaus Embgen

June 26, 2015

Abstract

Every program has its faults and every programmer makes mistakes in his code. What we need, is a system to find these mistakes and remove them from during development. Defensive programming gives us all the tools necessary to do just that. It helps you to make applications stable, reliable and to minimize user annoyance.

Introduction

The idea of defensive programming is based on defensive driving [**?**, p. 1]. That is to say that in defensive programming we are generally suspicious of everything that goes unchecked. So in the "cold, cruel world of invalid input" [**?**, p. 2] and buggy code we need a way to protect and defend ourselves and our code. This is where defensive programming comes in.
But protecting you is not all it is good for. Being defensive also helps making your program more polished and optimized. For example "a good program never puts out garbage" [**?**, p. 1], so the infamous "garbage in, garbage out" [**?**, p. 1] is not permitted.
In the following text I will explain what defensive programming is, how it is used and why it is viable.

What is defensive programming?

Defensive Programming is an important aspect of every qualitatively worthwhile product and should be considered while and after construction of the code. The topic of defensive programming is very relevant in times of increasing complexity and exploitation. Invalid input and bug-ridden code can

make a finished program unstable and in extreme cases unusable. With defensive programming you can design your program in such a way, that input is validated and code is bug free.
When used with other techniques [**?**, p. 2] defensive programming can be a powerful tool, to increase the value and quality and reusability of your code. If you are tasked with creating the OS of a medical radiation machine for example you have some high level decisions to make. Regarding defensive programming you ave to ask yourself how and where you want to process errors. This, in a practical sense, is the pivotal question of the whole procedure. Before implementing any error processing it and should be stood by at least for the debugging phase. What this comes down to is in essence producing correct or robust code.
If we look at the example with the radiation machine again, you can guess how important error handling and the style thereof can be. If the machine produces a wrong result in its output, a person could be hurt or worse. So rather than keeping the machine running in spite of errors you aim to shut it down completely.
What this comes down to is the decision between "correctness and robustness" [**?**, p. 12]. For a robust program one needs to use assertions to check for the error and then handle it if found with error handling [**?**, p. 7]. Robust code is not designed to have correct output but work reliably. This method finds widespread use in applications for consumers [**?**, p. 12].
For safety critical applications, code has to be correct [**?**, p. 12]. Wrong output is unacceptable and if detected the error will be handled in drastic ways. Correct code is designed to be factual rather than reliable.
Regardless of what method or form of defensive programming you choose for your code, beware of making your program transparent through error messages. Hackers can use error messages to learn about the inner workings of the code and attack it more easily. [**?**, p. 10]. The industry standard is to display something like "internal error" and a mail address or a telephone number to reach customer support.

How is defensive programming used?

There are three essential practices for defensive programming from which all others derive. Firstly you check the validity of external sources [**?**, p. 2], which is also a critical part of software security. Secondly you check the inputs of routines in your code [**?**, p. 2] to make sure your own code is bug free and for example type conversions are correctly executed. Thirdly and most importantly you have to decide how to handle bad in- and outputs,

which I already explained in "What is defensive programming".

Assertions

"An assertion is code that's used during development [...] that allows a program to check itself as it runs" [**?**, p. 3]. Assertions help by keeping the code correct if it is modified [**?**, p. 4] and check assumptions made on your part about certain parts of the program. You are encouraged to even use assertions on functions you trust [**?**, p. 12] to make sure that everything is actually the way it should be.
But one of the main duties of assertions is in fact checking the correctness of preconditions and postconditions [**?**, p. 6]. Preconditions being the conditions, that are promised to be true before a called routine. Whereas postconditions are the ones after the called routine.

Error Handling

"Assertions are used to handle errors that should never occur in the code. How to handle errors that you do expect to occur?" [**?**, p. 9]
I does not matter how bug free your code is. At the end of the day there are events that are simple out of your control. Most of the time this will happen if the user tries to load a corrupted or non-existent file or otherwise unusable input into the application.
So what does one do if wrong inputs are detected? There are several methods that you must weigh against each other. You could for example return a value that is known to be harmless. Or in code that is consecutively executed a lot you can just substitute the next piece of valid data [**?**, p. 9].
In error handling displaying a short message to the user is used for most things, e.g. "Server Error 500". The trick is to let the user know something happened, but you do not actually want people to have all the information about the error [**?**, p. 10].
The question you have to ask yourself now is where and how to handle your errors. If you handle every error locally, the performance of your code is going to suffer [**?**, p. 11]. As you can see, the decision how you process errors has to be a "high level design choice" [**?**, p. 12].

Exceptions

"I don't know what to do about this; I sure hope somebody else knows how to handle it" [**?**, p. 13]. Exceptions have something in common with inheritance. If used judiciously they reduce complexity, if used irrationally they

can make your code a nightmare to comprehend [**?**, p. 14].
As a rule of thumb you should use exceptions only for truly exceptional conditions [**?**, p. 14] to reduce your use of them and optimize the resulting code. Exceptions are also an invaluable source for information if used correctly. If you include all the information on why the exception was thrown [**?**, p. 16], fixing the code is made very easy.
While getting lazy when using exceptions, using empty catch blocks can be very tempting if you have not got an idea what to do. But such an approach implies that either the code in the try block is wrong and must be rewritten. It could also be that the catch block is wrong because it doesn't handle a valid exception in a correct way [**?**, p. 16].
To be consistent in your use of exceptions, and you should be [**?**, p. 18], you can consider building a centralized exception reporter. It contains all the possible exceptions that can occur, it handles each one the right way and controls the information that is given out [**?**, p. 17].

Barricades

Barricades can be imagined like the hull of a ship, in which certain parts can be sealed off in case of a hull breach [**?**, p. 19]. In computer science they work kind of like firewalls and they were actually called firewalls until it started referring only to port blocking [**?**, p. 19]. To compartmentalize your code into certain dirty and safe zones [**?**, p. 19] to reduce the need for checks in the whole program. If you take a routine that loads an external file into the program, it would classify as a dirty zone as you first have to validate the file before it is usable. So you build doors that have to be passed by parameters in order to be deemed clean and valid [**?**, p. 19]. And due to code deterioration doing several of these checks at strategic points in your code is encouraged.
External data should always be checked immediately [**?**, p. 20] for obvious security reasons. Converting data types as soon as possible reduces complexity and increases stability.
And this is the point where the distinction between assertions and error handling becomes very apparent. In dirty zones errors should be handled, because they are expected. Assertions are used in safe zones, because no errors should occur [**?**, p. 20].

Evaluation

With all this information and all these methods of writing good code, it is important not to forget that the development version does not equal the release version [**?**, p. 20]. Often programmers project the limitations of the release version, such as performance, look and text onto the development version which eliminates a lot of freedom to improve the code. In essence this means that in your development version you are free to have slow code [**?**, p. 20] full of detailed error messages and program ending assertions to flush out even the most hidden faults.
It can also be interesting to use offensive programming in your development. This basically means testing your program to the limit yourself by completely filling all allocated memory to detect allocation errors, making sure assertions abort the program [**?**, p. 21] and so on. This makes errors hard to overlook in your development process.

Disadvantages with defensive programming

As with all things you should be careful not to be overzealous with your error processing in the release version. If you check every nook and cranny in your code at all times, it is going to be really fat and slow [**?**, p. 26].
Having a good error documentation makes it easy to exploit your code and break your application [**?**, p. 2].
In applications like *Micrsoft Word*, with millions of lines of code and hundreds of modifications applied to the code, there is no way to check for every conceivable error, so error handling has to be implemented as an addition to assertions [**?**, p. 8].

Conclusion

So in conclusion there are some important points I want to emphasize:
Use assertions for your mistakes and error handling for the mistakes of others. This is of course just a rule of thumb.
Remove code that checks for trivial errors. It just slows down your code.
Leave code that checks for important errors.
Use a standardized approach to error handling.
Does your code avoid throwing exceptions in constructors and destructors? [**?**, p. 27]
Did you decide for robustness or correctness?

These are the most important guidelines to use defensive programming to your advantage.

References

[1] Steve McConnell *Code Complete 2*. Chapter 8. Defensive Programming, 1993.