Defensive Programming

Steven Bucksbaum, February 5, 2011

Promote Errors, Don't Bury Them

Defensive programming means raising errors loudly when they occur via testing or by throwing exceptions. The purpose, identify errors and issues, and either code for them or correct them.

Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software in spite of unforeseeable usage of said software. The idea can be viewed as reducing or eliminating the prospect of Murphy's Law having effect. **Defensive design** is the practice of anticipating all possible ways that an end-user could misuse a device, and designing the device so as to make such misuse impossible, or to minimise the negative consequences.

Exceptions are like accidents, and exception handling is like insurance. Exception handling is really no substitute for good defensive programming. Exceptions are "expensive" in terms of the resources and additional overhead they incur. Because of this, you should anticipate when, where, and how exceptions could be thrown, and do your best to "steer" your code around them.

The bottom line is that catching exceptions (not throwing them, but catching them) is expensive and if there are techniques that allow your program to continue functioning properly or otherwise gracefully handle exceptional conditions, known as "defensive programming", you should opt for that approach

Testing, defensive programming calls for testing early in the development process and test often. Find errors and deal with them before allowing customers to work the application. After the code is in production, monitor your site errors and act upon them

A short list of issues to check and code for:

1. Process external systems data properly
2. Handle unexpected conditions
3. Always validate input.
4. Casting, check to make sure you have the correct object
5. Using constants, ensuring constant correctness takes time.
6. Use enums where ever you have a series of discrete and possibly disjoint values.
7. Buffer checks: for virus prevention and memory security
8. Check function return values for errors
9. Anticipate exceptions, catch them and deal with them.