Invariants – a condition that can be relied upon to be true during execution of a specific part of an application. They include assumptions made about global or class level data.

To document invariants we use assertions.

Asserts – A warning to the developer. It complains loudly at runtime if an asserted assumption isn’t true.

System.Dignostic – contains the Debug class which has method:

Debug.Assert( bool, string); - many overloads. This version takes a bool as the assertion and a message if it fails.

Adds a few more lines of code. The Debug class is only executed in Debug runtime, not Release.

Assertion raises an exception.

Use assertions to document assumptions.

Anticipated issues and exceptions:

* Invalid user entry
* Invalid or missing data
* Code construct issues
* System issues

Invalid user input:

* use an appropriate control with restraints in value ranges or options.
* Use built in validation. Binding mechanisms can provide this
* Write validation methods for complicated validation.
* Use guard clauses
* Use a good default value to proceed. Will it provide predictable results?

Invalid or missing data:

* Validate incoming data.
* Proceed without the value
* Proceed with a good default value if it’s appropriate
* Display a message.

Code construct:

* Proceed with a default operation
* Ignore the issue. If it’s appropriate
* Log the issue and display a message

System issues:

* Try again
* Proceed with an alternative operation
* Ignore the issue if not critical
* If it needs to abort, log it using an exception and abort

Only notify the user when it is necessary

“To expect the unexpected shows an utterly modern intellect”

Global Exception Handler – the key to defending your code against unexpected exceptions.

Look up global exception handler for your desired platform or application.

Add the global exception handler first.

Win forms recommended global exception handler:

//for UI thread exceptions

Application.ThreadException += new ThreadExceptionEventHandler(GlobalExceptionHandler)

//force all windows forms to go through our handler

Application.SetUnhandledExceptionMode(UnhandledExceptionMode.CatchExecption)

//for non-UI exceptions

AppDomain.CurrentDomain.UnhandledException += new UnhandledExceptionEventHandler(GlobalExceptionHandler)

We don’t want the app to continue.

Log the issue and then exit.

When we throw exceptions, if we don’t try and catch then the global exception handler catches it.

This isn’t good if you don’t want the app to terminate.

Handling anticipated exceptions:

* Don’t use the GEH
* Those that are explicitly thrown or anticipated in the code
* Handle them locally in time and space. In time means as soon as possible after it’s thrown. In space means close to where it’s thrown. The closer it is, the more info available
* Try{} catch{} – inside the try is the guarded code which Can throw an exception
* Throw on it’s own will pass the exception up the global call chain.
* Do something. Never catch and do nothing. Never catch unless you want to do something.
* Catch specific exceptions. Because that’s what we expect. Whats anticipated.
* Log. Collect important info about the operation of the application. Log if the user can’t do anything about the error. Not always necessary.
* Pokemon exception handling – gotta catch em all. Bad practice. Code bloat, inability to catch specific errors.
* Only catch anticipated errors and let the global handler catch the rest.

Throw ex – throws a new exception rather than throwing the original exception. Tis will lose some of the original information from the originating exception. Best practise is to throw.