Visual Studio

**Unit Testing**

The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect. Each unit is tested separately before integrating them into modules to test the interfaces between modules. Unit testing has proven its value in that a large percentage of defects are identified during its use.

The most common approach to unit testing requires drivers and stubs to be written. The driver simulates a calling unit and the stub simulates a called unit. The investment of developer time in this activity sometimes results in demoting unit testing to a lower level of priority and that is almost always a mistake. Even though the drivers and stubs cost time and money, unit testing provides some undeniable advantages. It allows for automation of the testing process, reduces difficulties of discovering errors contained in more complex pieces of the application, and test coverage is often enhanced because attention is given to each unit.

For example, if you have two units and decide it would be more cost effective to glue them together and initially test them as an integrated unit, an error could occur in a variety of places:

* Is the error due to a defect in unit 1?
* Is the error due to a defect in unit 2?
* Is the error due to defects in both units?
* Is the error due to a defect in the interface between the units?
* Is the error due to a defect in the test?

Finding the error (or errors) in the integrated module is much more complicated than first isolating the units, testing each, then integrating them and testing the whole.

Drivers and stubs can be reused so the constant changes that occur during the development cycle can be retested frequently without writing large amounts of additional test code. In effect, this reduces the cost of writing the drivers and stubs on a per-use basis and the cost of retesting is better controlled.

Visual Studio

**Integration Testing**

Integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested. A component, in this sense, refers to an integrated aggregate of more than one unit. In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program. The idea is to test combinations of pieces and eventually expand the process to test your modules with those of other groups. Eventually all the modules making up a process are tested together. Beyond that, if the program is composed of more than one process, they should be tested in pairs rather than all at once.

Integration testing identifies problems that occur when units are combined. By using a test plan that requires you to test each unit and ensure the viability of each before combining units, you know that any errors discovered when combining units are likely related to the interface between units. This method reduces the number of possibilities to a far simpler level of analysis.

You can do integration testing in a variety of ways but the following are three common strategies:

* The top-down approach to integration testing requires the highest-level modules be test and integrated first. This allows high-level logic and data flow to be tested early in the process and it tends to minimize the need for drivers. However, the need for stubs complicates test management and low-level utilities are tested relatively late in the development cycle. Another disadvantage of top-down integration testing is its poor support for early release of limited functionality.
* The bottom-up approach requires the lowest-level units be tested and integrated first. These units are frequently referred to as utility modules. By using this approach, utility modules are tested early in the development process and the need for stubs is minimized. The downside, however, is that the need for drivers complicates test management and high-level logic and data flow are tested late. Like the top-down approach, the bottom-up approach also provides poor support for early release of limited functionality.
* The third approach, sometimes referred to as the umbrella approach, requires testing along functional data and control-flow paths. First, the inputs for functions are integrated in the bottom-up pattern discussed above. The outputs for each function are then integrated in the top-down manner. The primary advantage of this approach is the degree of support for early release of limited functionality. It also helps minimize the need for stubs and drivers. The potential weaknesses of this approach are significant, however, in that it can be less systematic than the other two approaches, leading to the need for more regression testing.

Visual Studio

**Regression Testing**

Any time you modify an implementation within a program, you should also do regression testing. You can do so by rerunning existing tests against the modified code to determine whether the changes break anything that worked prior to the change and by writing new tests where necessary. Adequate coverage without wasting time should be a primary consideration when conducting regression tests. Try to spend as little time as possible doing regression testing without reducing the probability that you will detect new failures in old, already tested code.

Some strategies and factors to consider during this process include the following:

* Test fixed bugs promptly. The programmer might have handled the symptoms but not have gotten to the underlying cause.
* Watch for side effects of fixes. The bug itself might be fixed but the fix might create other bugs.
* Write a regression test for each bug fixed.
* If two or more tests are similar, determine which is less effective and get rid of it.
* Identify tests that the program consistently passes and archive them.
* Focus on functional issues, not those related to design.
* Make changes (small and large) to data and find any resulting corruption.
* Trace the effects of the changes on program memory.

**Building a Library**

The most effective approach to regression testing is based on developing a library of tests made up of a standard battery of test cases that can be run every time you build a new version of the program. The most difficult aspect involved in building a library of test cases is determining which test cases to include. The most common suggestion from authorities in the field of software testing is to avoid spending excessive amounts of time trying to decide and err on the side of caution. Automated tests, as well as test cases involving boundary conditions and timing almost definitely belong in your library. Some software development companies include only tests that have actually found bugs. The problem with that rationale is that the particular bug may have been found and fixed in the distant past.

Periodically review the regression test library to eliminate redundant or unnecessary tests. Do this about every third testing cycle. Duplication is quite common when more than one person is writing test code. An example that causes this problem is the concentration of tests that often develop when a bug or variants of it are particularly persistent and are present across many cycles of testing. Numerous tests might be written and added to the regression test library. These multiple tests are useful for fixing the bug, but when all traces of the bug and its variants are eliminated from the program, select the best of the tests associated with the bug and remove the rest from the library.