Designing Software for Testability

Alfred J. Barchi
ajb@ajbinc.net
http://www.ajbinc.net/

Abstract

As software-based systems and applications become increasingly more complex, there is a tendency for them to become increasingly fragile and less stable. One obvious reason for this is that the bigger and more complex a software application becomes, the more difficult it becomes to adequately test. Currently available commercial software testing products attempt to address this by replacing the user I/O functions (keyboard, mouse, display window, etc.) with a script-driven test manager that exercises the software product. This approach suffers from the problem that software is seldom designed to work in this type of environment, and the number of ways that a software product can interact with its environment is relatively unconstrained, whereas the capabilities of the automated test tools aren't.

This paper outlines an architectural approach to designing software that is amenable to being tested with automated test tools. With this approach, commercial test suites can be combined with engineering software written specifically to test a software product, in order to provide a more effective means of performance testing, regression testing and stability testing.

In the past, developers and managers have argued that adding functionality that does not support the mission of the software is an expense that is not justified, either by customer requirements or commercial value. The author makes that case that designing for testability is always justified, both by the need to adequately verify that requirements are being met and by the commercial advantage of producing a more stable, error-free product. This increases customer acceptance and reduces maintenance costs. Additionally, the author argues that an architecture that supports testability is inherently more adaptable to new uses, and ultimately saves future development costs.

The Problem

The problem, simply stated, is that software development organizations have a tendency to develop products that are difficult to test. They don't make the effort to design in features to avoid this problem. There are a variety of reasons offered for this: "It'll cost too much money!" "It'll take to much time!" "It's too difficult!" "It's not in the spec!" "We'll just use X-Runner (or Win-Runner, or Rational Robot, etc.)." "It doesn't add any value to the final product."

I've heard all of these reasons and more, over the years, coming from some highly intelligent and capable software engineers. In addition, engineers and engineering managers have a tendency to equate the verification and validation process with the testing process. This last fallacy has even been institutionalized in such engineering-process/baseline-management standards as DoD-Std-2167A and Mil-Std-498, both of which cover the V&V process, performance testing to verify performance requirements, etc., but do not contain separate sections for stability testing, ensuring that the product handles inputs correctly over the entire range of allowable values, processes invalid inputs correctly, and functions correctly in contended environments (i.e., on a host computer where there are other, unrelated processes running simultaneously and using system resources, such as on a server). Basically, if you want your software product to work correctly over its entire range of inputs, you have to write a requirement that says this and then verify the requirement!

One might expect that a systems engineering department full of engineers raised on a diet of black-boxes, feedback control loops and the theory of control systems would tend to view the software development process as a closed-loop control system with testing providing the feedback path necessary to produce high quality results. Well, they don't (I was once asked to remove the word "hysteresis" from a presentation I was preparing to give because none of our "engineers" knew what it meant). You can draw all of the diagrams of the software development process that you want, you can show your management that you have, at least in theory, a process that will produce good results, but unless you understand what testing really means in such a system, your results will be less than spectacular, i.e., cost overruns, schedule slips, serious design flaws, instability and buggy behavior of your final product, etc. Testing is more than just a box on a process diagram, and you need to have a thorough understanding of what that box represents.

"It'll cost too much money!", "It'll take to much time!" and "It's not in the spec!" – What does this really mean? The person saying this generally means that his existing design concept doesn't support testing very well and he perceives that making his design testable will require additional features that nobody planned for when the original proposal was submitted (whether for an in-house development or as an external contract). Unfortunately, from his perspective, he is correct, and it will be very difficult to change his thinking.

Yes, testing costs more than not testing (at least initially). Yes, developing automated tests means you have to write software, whether you are using a tool such as X-Runner or writing your own test software from scratch. Yes, fixing the problems that testing surfaces takes time and can jeopardize your delivery schedule. However, designing your product in such a way that you can test it with automated test tools does *not* mean that you have to write more code, merely that you have to organize things somewhat differently.

"It's too difficult!" and "It doesn't add any value to the final product." – The person who says this really means that he doesn't understand the testing process, doesn't

understand the concepts of flexibility and adaptability, and would just as soon not be bothered (I once had a high-level executive respond to my assertion that we needed to test our products better before we shipped them, “Al, our customer has a whole department full of people spending 8 hours a day trying to get work done on our product. They’ll find our bugs a lot faster than you ever will through in-house testing, and they’ll call us when they do. And what’s more, we can generally get them to pay us to fix our problems!” Of course, we had difficulty getting any new development work done because our engineers were constantly at customer sites all over the country debugging software problems, and we had to use our in-house computers like a travel agency, and we also got threatened with lawsuits a lot. The problem (aside from the obvious ethical issue) with this approach is that detecting a bug is considerably easier sometimes than diagnosing and fixing it. You still have to spend considerable time in some cases to develop test software to help you debug a problem. Not only that, but your customers might not recognize anomalous behavior as a problem until it results in some secondary loss or damage.

The fact is that it’s not too difficult to make software amenable to automated testing, if you understand the principles involved. In addition, the architecture required to accomplish this has the added benefits of making your product more “flexible” (i.e., easier to migrate to newer technologies as they become available), and more “adaptable” (i.e., easier to adapt to new uses that were not part of the original purpose for making the product). These last two benefits ultimately lead to the opportunity to generate higher revenues for less cost.

“We’ll just use X-Runner (or Win-Runner, or Rational Robot, etc.).” – Someone who says this simply doesn’t understand the nature of the problem. Automated test tools such as these are typically excellent, well-thought-out products, but they do have their limitations.

First, they’re slow. They typically work by intercepting calls to and returns from the presentation layer (X-Windows, MS-Windows, MFC, etc.). Operations such as keystrokes, mouse-clicks, etc. are recorded as a script that is later played back through an interpreter to duplicate these operations. Behavior of the product-under-test is compared to recorded values that were returned when the script was recorded. Any interpretation of these values generally requires a developer to write additional code in the script language. This code is also executed by the interpreter, which further slows down the process.

Second, they place their own significant limitations on the architecture of the system. In order for your test tool to understand window objects such as frames, buttons, edit boxes, etc., and not just pixels, you have to design your HMI interface in a specific way, or the test tool cannot interface with it. The APIs for X-Windows, MS-Windows and Microsoft Foundation Class are vast and complex, and it is beyond the ability of anyone to anticipate all of the variations possible when using these frameworks, particularly when developers start using undocumented features (“MFC Internals” by Shepard and Wingo is an excellent, 700 page example of how to do exactly this with MFC).

Third, the functionality that you typically need to test is the stuff that lies behind the human-machine interface, not the interface itself (you have to test that too, but that's a topic for a different paper). In order to do this in an automated way, you have to create test software with the ability to analyze outputs over a wide variety of inputs. This is difficult to do efficiently in the scripting languages these tools provide, and if you do it externally to the test tool, then you will discover that interfacing your analysis software to the test tool is problematic.

Fourth, these tools are typically not what you would want to use for detecting stability problems, such as memory leaks, boundary-limit problems (e.g., writing past the end of a queue), race conditions and deadlocks. The reason for this is that the tools are large, slow and cumbersome and they affect the underlying timing of the software being tested. To detect stability problems, particularly in a client-server architecture, you need to run lots of clients simultaneously under heavy loads in order to manifest the problems in an efficient manner. If I need to run 30 clients to do this, for example, and I can only run 5 because of the limitations of my test tool, I have a problem.

The good news is that if you design your product correctly, you don't even need these tools for testing your core functionality (you still might need them for testing the HMI).

The next section discusses the general design principles necessary to make software amenable to automated testing. Following this, there is a section that uses a concrete example to illustrate how these principles are applied. The example itself is trivial, the discussion of the application of the design principles is not. After that, there is a discussion of the secondary benefits of applying these design principles effectively, followed by a closing summary of what was discussed.

General Design Principles

This section describes the principles involved in designing software for testability. Some of this may sound pretty trivial and basic. Please bear with me. Think of it the same way you would think of the fundamental principle for making money in the stock market: "Buy low, sell high." As simple as it sounds, a lot of intelligent people lose money in the stock market every day, because they buy high and sell low.

#1: Identify the core functionality – This is not always easy to do. For example, consider a program like Quicken or Microsoft Money. Is the portion of the software that displays an account register and accepts new entries, deletions and changes part of the core functionality? Or, how about a sound wave editor? Is the portion of the software that displays the wave in a window and accepts edits to the wave part of the core functionality?

The answer is that the core functionality is that which gives your software its identity. Displaying a window on a graphics display is part of the core functionality of X-Windows, or MS-Windows or whatever other framework you use to write to a graphics display. The same is true for accepting keystroke and mouse inputs. In the examples above, the software that interprets keystroke and mouse inputs passed to it from the window-management system is core functionality, as is the software that maintains an internal representation of the register or the sound wave and knows how to draw them, but not the software which actually does the drawing.

This is an important distinction, because when you define the interface to your core functionality properly, it allows you to design automated tests to determine such things as whether it works over its full range of valid inputs, whether it works correctly when subjected to a wide range of invalid inputs, whether it is thread-safe, etc. It also allows you to re-use such tests as part of a regression-testing program.

#2: Encapsulate the core functionality – No, this is not going to be a discussion of the merits of object-oriented programming. All I mean here is that you need to identify the elements of your software that you want to test with automated and isolate them behind a call-level interface. Consider the following test for encapsulation:

1. Can I compile the software that defines my core functionality in a separate module?

2. Can I call my core software functions from a different programming language?

3. Can I migrate my core software to a new host or programming environment (e.g., migrate from MS-Windows to Unix)?

4. Can I exercise all of the functions and set all of the options and parameters of my core software via my call-level interface?

5. Can I bypass my call-level interface?

Some proponents of object-oriented programming will likely proclaim that when you code your software using an object-oriented language such as C++ or Smalltalk, you automatically achieve encapsulation, since encapsulation is one of the principles on which object-oriented programming is based and one of the basic features of the definition of object classes. Well … it depends on what you mean by “encapsulation.” Or, to put it another way, “How ‘black’ is my black-box?” Consider items #1 and #2 above. If I define an object class using C++ and then publish a definition of the public portions of it in a header file, I can certainly compile it as a separate object module that is later linked to the main program when the program is built. But, can I write the main program in any program other than C++? Well, you might want to do this if you want to write your test software in Visual Basic, Java or Rational Robot’s script language, for example.

The fact is that there are several issues that you have you deal with whenever you program in a language such as C++ which uses strong data-typing and other abstractions such as object classes. First, if you define an object class interface, you also have to include some method for instantiating the object. This means that the language that your test software uses must also support this, and in the same way. Second, strongly typed languages typically require a support environment that includes exception handlers. This means that you don't just write stand-alone modules in C++ or Ada, for example, and expect to successfully link them into a C, Fortran or Basic main program. Third, when you mix languages, you always have to deal with inconsistencies in the internal representation of data structures.

Item #1 is also important as a mental test to verify that you have a clear understanding of just what your core functionality really is.

Item #3 is a test of how divorced your software is from the underlying environment. For example, if you use MFC string manipulation functions, you're going to have a difficult time migrating to a Unix platform – this isn't necessarily a bad thing, just something to be keep in mind. You are also going to be exercising MFC string manipulation functions every time you test your software. You have to be aware of this whenever you re-compile your software with an updated version of MFC. Using threads is another potential source of problems. Different thread libraries tend to function slightly differently and may contain bugs. You not only need to test for these problems, but you need to design your test software in such a way that it can migrate along with your core software. Item #3 may not really be as germane to testing as some of the other aspects of encapsulation, but it is important to the bottom line.

Item #4 is a key point to remember when you design your core software. The goal that you want to achieve with automated testing is a stand-alone suite of tests that can run overnight, unattended. If manual intervention is required to change settings or parameters, you will obviously have a much more difficult time achieving this goal.

As far as item #5, if you can bypass your API, then there is another interface that you have to test. You also have to test for unexpected problems arising when you attempt to use both interfaces simultaneously (particularly if your core software has states or uses multi-call transaction sequences).

The reason that I recommend a call-level interface is that it is simple and you can use it no matter how complex your core software is. Basically, you make a call to a single interface function and pass it a pointer to a data structure that contains numerical value that represents the operation to be performed as well as any other information required by the operation. The interface function executes a 'case' statement to call the function that performs the specific operation being requested. Return data is passed back the same way, and can even be passed back in the same data structure, depending on what operation is being performed.

Not only does virtually every programming language in existence support this kind of interface, but you can build a simple front end which allows it to run as a stand-alone module. This is significant when you consider that some automated test tools, such as early versions of X-Runner, do not allow you to link pre-compiled object modules directly to their internal test scripts. The test scripts are executed by an interpreter internal to the test tool – you would have to link your pre-compiled module directly to this, and then have some mechanism built into the test script language to allow you to invoke your pre-compiled module, all of which imposes significant constraints on the kind of interface you can use and tends to discourage test tool vendors from even making the attempt to provide this kind of functionality. This is a similar situation to the one addressed by the Java Native Interface (JNI). With JNI you generate special header files for your application and compile and load it as a shared library in order to call it from within Java – this is an example of a significant constraint.

#3: The kind of functionality that you want to test helps define your API – The example that I will discuss in the next section provides a good illustration of this. Basically, when you pass data between your core software and your external interfaces, you want to minimize the amount of translation which takes place, especially when you are mixing programming languages or targeting multiple host environments. Additionally, you need to consider whether there is any data, such as state information, that is being used within your core software that would facilitate testing if you were to expose it via your API. And, finally, you need to consider whether there is any additional data, such as time-tags, which it would be useful for your software to collect and pass to the test software, any additional data that it would be useful for your test software to send to your core software, and any additional functionality that your core software needs to provide to facilitate testing.

#4: You shouldn't have to add extra functionality or write extra software just to support testing – Your test software should treat your core software as a black box. It needs sufficient control via your API to completely exercise all of the functionality of this black box. If you find yourself adding additional functions or data to the API in order to accomplish this, it usually means that there are other requirements on your design that you haven't discovered yet and which will become manifest later on in the design process.

That's it. The principles are simple. They are easy to apply, in theory. But as Yogi Berra once said, "In theory, there is no difference between theory and practice … in practice there is." The next section uses an example to discuss how to apply these principles.

Discussion by Example

As they say in the chess world, there is theory and then there is praxis. Anyone who has ever read a chess book knows that even the most theoretical of them rely heavily on examples to demonstrate how principles are applied in real-life situations. The reason for this is that theoretical principles cannot simply be applied blindly or in a mechanical fashion. In chess, although there may be many situations with similar characteristics, each different position is also unique in some way, and abstract principles must be weighed and balanced against one-another and tailored to the immediate situation. It is this ability to place principles in perspective and exploit the unique characteristics of any given situation that separates the master from the amateur. And so here is an example to help illustrate the application of the principles of designing software for testability. The example itself is a trivial application, easy to implement in a single afternoon. Nevertheless, the same design principles still apply, regardless of how small and simple or how big and complex the project.

Many years ago, when I was shopping for my first house, I wanted to compare mortgage rates and different options for length of loans and early payoffs. So I wrote a set of calculators. One computed the size of a payment given the initial loan size, interest rate and term. Another one based on the same algorithm printed out a complete amortization schedule. A third one computed the term required to amortize a loan given the principal, interest rate and size of the payment. A fourth calculator determined the interest rate from the principal, payment amount and number of payments.

Over the years I have implemented these calculators in C, Basic and even in spreadsheets. For this paper I decided to implement all four in a single application, written in C++ and using a modeless dialog box with the Microsoft Foundation Class (MFC) library. Since I already had these calculators written in C, I was able to reuse most of this code.

The dialog box would have edit boxes for each value: principal, yearly interest rate, number of payments per year, total number of payments, payment size, total interest paid and total amount paid. The amortization schedule would be displayed in a pop-up text edit window, and would utilize the built-in capabilities of MFC to edit it and save it as a text file. The dialog box would contain command buttons to execute each type of calculation as well as a ‘help’ button and an ‘exit’ button. It would also have a pop-up modal dialog window that would display ‘about’ information (version number, copyright notice, etc.). The ‘help’, ‘exit’ and ‘about’ functions are provided as standard features by MFC.

The first thing to consider was the question of just exactly what was the core functionality of this application. The first pass at this was easy, particularly since I already had most of this code written. My core functionality consisted of all the functions necessary to perform financial calculations and generate amortization tables. As the design matured, I would re-visit this question a couple of times, at least partly because of the vagaries of MFC.

The second question was also easy, at least at a high level – how do I encapsulate my core functionality. Since, as I stated in the previous section, I am fond of doing this by implementing a single API control function that is called with a pointer to a structure containing the operation to be performed along with necessary data, this is what I did. Initially, the structure looked something like this:

```
typedef struct finance {
   unsigned short int  command ;
   double              principal ;
```

```
    double              yearly_interest_rate ;
    double              periods_per_year ;
    unsigned long int   term ;
    double              payment ;
    double              total_interest ;
    double              final_value ;
    char               *amortization_table ;
} *finance_data ;
```

The actual parameter that is passed to my API function is a pointer of type ‘finance_data’ which points to a structure that can either be static or dynamically allocated. The amortization table is generated using a string buffer that is dynamically allocated and resized upon completion of the table so that it matches the size of the table. The table itself is a single long text string.

All of the values, including result values as well as inputs provided by the user are included in this single structure. There were two reasons for this. First, I wanted to have a single data block that contained all of the information that would be displayed in the dialog box. And second, some of the computed values could actually be used as inputs or fed back in to the other types of calculations. Placing all of the values in a single data block meant that I didn’t have to move data around to accomplish this.

The next task was to design the user interface. The Microsoft Development Studio makes this easy. I specified the kind of application that I wanted to create and it generated all of the skeleton code necessary to do this. I then laid out how I wanted the dialog box to look and tied the buttons and edit windows to MFC message handlers that called my core software. All of this took very little time. Initially I used the standard DoDataExchange function provided by MFC along with the appropriate DDX and DDV calls to load, retrieve and validate data in the dialog boxes.

Unfortunately, I wasn’t satisfied with the behavior of my application when I used the standard mechanisms provided by MFC. When you use the DDX functions, MFC passes the entire text string in the edit box. Unfortunately, when you send a string to an edit box, MFC places the curser to the left (or beginning) of the string, rather than to the right. When you use the DDV functions, if an illegal value has been entered, the standard validation function pops up a modal dialog window that tells you that you just entered an illegal value and highlights the entire string in the edit box. The DoDataExchange function that calls the DDX and DDV functions is only called when you change the focus from the current dialog control (e.g., the edit box) to a different one.

What all of this means is that using the standard mechanisms of MFC, my application would allow a user to enter any number of illegal characters into an edit box and then prompt him to edit or re-enter the entire value when he is done. This was unsatisfactory to me. I wanted my application to validate keystroke entries on the fly and simply not allow a user to enter an illegal value.

I couldn’t simply set up a message handler that got called whenever a new character was entered into an edit box, validate the string, correct it and write out the validated string back to the edit box because of the problem with the cursor changing position (I wanted the cursor to remain on the right end of the edit box after making a correction). I also couldn't do this by calling DoDataExchange from the OnChangeEdit function because this sets up an infinite loop which locks up the computer (DoDataExchange causes a change to the edit box which then invokes the OnChangeEdit function which calls DoDataExchange …). Argghhh!

I discovered that I could validate the last character entered by fetching the entire string from the edit box window while in the OnChangeEdit function, validate the last character entered using my own validation code (remember, the DDV functions don't behave the way I want them to and cannot be called from the OnChangeEdit function anyway), and if it is an illegal character, simply send a backspace character to the edit box window using the a 'WM_CHAR' message. As long as I was careful to deal with the special case of the first character to be entered in the edit box (MFC calls itself recursively when you send a window message, which causes OnChangeEdit to be called again with a zero-length string when you backspace the first character), I was OK and I could obtain the behavior I wanted. (As a side note, I don't currently have access to Win-Runner or Rational Robot, and I have no way of knowing whether these tools would correctly handle this kind of behavior.)

So I ended up writing my own input validation functions. Are these functions part of my application's core functionality? The simple answer is 'yes'. The financial calculation functions expect to receive only valid inputs. The validation functions not only contain the definition of a valid input for this application, but they are essential to ensure the correct operation of the calculation functions.

So how do I encapsulate this functionality? Simple, I use the same call-level interface and pass a different data structure, which contains the command code to invoke the appropriate validation function as the first value, and the string data to be validated as the second value. Since different fields had different allowable values, I had to use a different command code for each different edit box string type to be validated.

My data structure now looked something like this:

```
typedef struct finance {

   unsigned short int  command ;

   union {

      struct finance_values {

         double              principal ;
         double              yearly_interest_rate ;
         double              periods_per_year ;
         unsigned long int   term ;
         double              payment ;
         double              total_interest ;
         double              final_value ;
```

```
            char               *amortization_table ;

         }  data ;

         struct valid_string {

            char               *validation_string ;
            unsigned short int  valid_string_flag ;

         }  validation ;

      }
   } *finance_data ;
```

It’s getting messier but it’s still manageable.

Now it was time to apply principle #3. What behavior did I want to test? Well, for starters, I wanted to make sure that it computes the correct values for its inputs. This is hardly the type of testing that would drive me to develop an automated test, although I might consider this approach for the purpose of regression testing if my application were large and complex.

As it turns out, even a trivial example like this can pose some interesting problems for which automated testing would be appropriate. In this case, the values that are computed by the different operations all achieve their results somewhat differently. The payment function computes the size of the payment using a compound interest formula. The “term” computation is done by running a loop that computes current interest and principal based on outstanding balance and deducts the principal from the balance until the balance is reduced to zero. The interest rate is computed using a binary search algorithm. These different methods result in slightly different final values due to differences in round-off methods, etc. When I was running these computations one at a time from a command prompt these differences were mildly annoying but of no great significance. However, with the new application, you can sit there and click on the different buttons repeatedly, and the numbers will constantly change. Eventually, after a number of operations, the changes reach the point where the total number of payments changes, causing a major change in the other numbers. This kind of behavior is totally unacceptable.

The way to correct the problem is to adjust the way each algorithm does its round-off so that they all produce the same outputs over a wide range of input values. But how do you know when your tweaking has been successful? The answer is to write a front-end test module that exercises the application over a wide range of inputs for each operation and compares the input to the output values. This would be easy enough to do except for the fact that round-off can occur when the floating point numerical values are converted back and forth between text-string and internal binary floating-point format. This was happening whenever a value was read in from or written out to an edit box.

It became clear that since I was concerned about the actual content of the edit box strings themselves, the software that generated and translated these strings would need to be part of the core functionality. The API would have to pass actual string values so that the conversions could be controlled within the core software. This actually led to a simplification of the API data structure:

```
typedef struct finance {
   unsigned short int  command ;
   unsigned short int  valid_string_flag ;
   char               *principal ;
   char               *yearly_interest_rate ;
   char               *periods_per_year ;
   char               *term ;
   char               *payment ;
   char               *total_interest ;
   char               *final_value ;
   char               *amortization_table ;
} *finance_data ;
```

Simple. Straightforward. Elegant.

Now I could not only develop a simple test driver to validate any tweaks I made to the algorithms, but the core software now had total control over the exact content of the edit boxes. As for design principle #4, no extra code was written to accommodate testing (OK, so an API control function was added – basically a large case statement – but I would have done this anyway, for other reasons which I'll discuss in the next section). All that was required was to account for the need for automated testing while organizing the structure of the application.

Although this is a trivial example that took longer to write about than it did to design, this same approach can be applied to even really huge designs with millions of lines of code. The interfaces can get considerably more complex, and there can be considerably more of them, but the same underlying principles apply.

Beneficial Side-Effects

Essentially, this paper describes an approach for developing software that encapsulates it within a black box that is independent of the specific host/platform interface being used to control it, Connects the host/platform to it through a call-level API, and utilizes this API as a way to connect to automated test drivers. What are some of the beneficial side-effects of this approach?

1. *Flexibility* – I define flexibility as "Ease of migration of the product to newer technology as it becomes available." This approach provides a certain amount of isolation for the core functionality from its host environment. In the example described in the previous section, for instance, I could implement the front end with Visual Basic, or X-Windows, or Java (using JNI), without changing the core functionality at all (I might have to re-compile it). This means I can port it to

Linux, VMS, or the MacIntosh, among others, by replacing only the front end.

2. *Adaptability* – I define adaptability as “containing design features that are necessary to support tailoring the product to uses other than those of the original application.” I can embed this application within a larger application (for example, if I wanted to build my own version of Quicken). I can add software to turn it in to a stand-alone server that is accessed over a network. I can make it in to a shared library (operating system permitting) that allows its functionality to be accessed by other applications running in the host environment.

3. *Cost* – I can create scaled-down versions of the application by limiting the allowable operations at the API (thereby providing an opportunity to sell low-end versions of the application at reduced cost, which could expand the market for the application). Additionally, the use of automated test drivers can greatly facilitate debugging activities, ultimately leading to reduced development and maintenance costs.

Of course, the main benefit of this approach is that it can dramatically improve the ability to test the application with automated test tools. This, in turn, leads to improved stability of the product (assuming you actually do stability testing), which produces the potential for greater customer and user acceptance.

Conclusion

Thorough testing is a fundamental part of the development process. It is one of the feedback paths that drive changes in system requirements, changes in design and changes in implementation. It’s an essential element of a product’s lifecycle. Every product that is developed and fielded is ultimately tested, either by the developer or by the developer’s customers. And if your customer tests your product and finds it lacking, you might not get paid at all.

Nevertheless, it has been a constant source of amazement to me over the years that otherwise intelligent people seem to consider testing as an “add-on” activity which ultimately gets cut back to reduce costs or meet delivery schedules. This same attitude causes these people to ignore the need to design products in such a way that they can be easily tested. They say things like, “We can’t afford it”, or, “It’s not in the spec” or, “We aren’t building a Cadillac!”

“Pay me now or pay me later!” – It doesn’t really matter whether you cut your testing program short to meet schedule or budget commitments. It doesn’t really matter that you may be able to pass the costs of fixing your product’s deficiencies on to your customers. Sooner or later, you’re going to have to diagnose and fix those deficiencies. If you haven’t designed the product in such a way as to facilitate this, it will simply take longer and cost more, and ultimately you will end up adding on features to your product

anyway, in order to do diagnostic testing. The end result of this kind of design cycle is never as satisfactory as doing the job right on the first pass.

Furthermore, your customers' satisfaction is inversely related to how much it costs them and how long they have to wait for you to fix the problems with your product that are depriving them of the full benefits for which they are paying. And unhappy customers can have a nasty way of paying you back.

As was discussed in this paper, designing software for testability requires careful thought. It requires the application of some basic architectural principles in a way that is complementary to other design objectives. It does not require extra software to be developed. It does require that the software that is developed be organized in a certain way. The result is not only a product that is testable, but also a product that is flexible and adaptable. The result is a product that is more stable and which has lower maintenance costs. The result is a better product with a lower bottom-line cost.