

Designing Software for Testability
By Jeffrey Feldstein
jbf@sherpas.com
<http://www.sherpas.com>

Copyright 2005-2006 Jeffrey Feldstein

When does software test begin? Many developers believe testing begins when all the features are complete and they hand off their work to the test team. While this may be what actually occurs in many development projects, it is far from the ideal. Software quality assurance begins in the definition phase of the project. One important aspect of software quality to consider is testability of the software architecture and design.

The testability of the application can have a profound effect on its overall quality. The more testable the software, the easier it will be to identify, isolate, resolve, re-test and verify the resolution of a bug. It is a well established fact that the sooner a bug is identified and fixed the cheaper it is to handle. In addition, a testable architecture allows some testing to occur at the component level. Bugs in smaller portions of the application are easier to isolate, repair, and verify. In general, testable architectures also allow for a greater degree of automation. More test cases can be automated and instead of just automating only the simplest test cases, complex scenarios, implemented with more robust automation is possible.

Typically test groups first see the product when the development team thinks it's finished and they "throw" the software "over the wall" to the test group. The test group will start up the system and pretend they are either a dumb or sophisticated user to try to break the system. If the test team did some prior work, they might have a test plan, which details all of the steps that they will perform to try to break the system. Testers that have been doing this for awhile might have gotten bored by manually repeating the same scenarios over and over and will begin to automate their work with the use of either a scripting package or a GUI test tool. Both of these approaches simply repeat the key-strokes, mouse clicks, or entered commands automatically, possibly at a speed faster than the human tester. By simulating a user sitting in front of the computer and attempting to check for correct responses from the software some bugs can be found or regressions checked. This type of automation usually occurs in an environment where the feature is first checked by hand, and once it's working, the automation is put in place as a "regression test".

There are several problems to this approach to automation. First, since you are testing the entire system together it can be difficult or tricky to isolate a bug once it's identified. It could be anywhere in the system. While finding the cause and fixing the bug might be the developer's problem and not our problem as testers, the longer it takes the developer to find the bug the more expensive it becomes to the project and schedule delays are more likely. Second, simulating a user at a keyboard does not lend itself to important types of system testing, such as stress, soak and reliability testing. Yes, an attempt can be made to run these scripts on many computers to simulate many users, but this setup is tricky to keep running. Using a load testing tool is another approach but these tools also test

through the front end, most of the time. In addition, simulating an end-user may not stress all the back-end components of the system such as accessing or updating a large database or generating all the possible error conditions such as several users attempting to update the same record at the same time. Third, it can be difficult to fully test the functionality of back-end system components strictly through the user interface. For instance, how can we be sure that resource scarcity such as low-memory, low-disk, slow network-bandwidth are fully tested entirely through a user interface. There is a lot of software between the user interface and handling of back-end resources and finding the right cause and effect to test these lines of code through the user interface can be tricky. This is why it's important to be able to write automation that is able to fully test and stress portions of the system in isolation. Testing components in isolation requires both a testable architecture and a test engineer with a good understanding of software engineering.

Testable architectures have another important benefit. When I first started automating software testing with a GUI test tool, I (and my fellow test engineers) spent a great deal of energy trying to work around the problems that automation caused, which zapped energy that could have been used to designing and implementing superior, more complete, and more deadly test cases. After all, what's more satisfying to a test engineer than implementing those killer test cases which reveal the truly nasty bugs? If we spend most of our energy figuring out how to automate simple test cases, where is the creativity reserve for our best tests? This same energy-zapping poor design can happen through other types of testing, such as testing solely through a command line interface (CLI).

Great, so now that you want a testable architecture you are probably wondering, "Gosh, on top of all of the problems on a developer's mind: building the product, an impossibly aggressive schedule, the Diet Dr. Pepper shortage in the cooler, how am I going to convince him to build a testable architecture?" Well I can't tell you how to guarantee that there is enough sugar-free caffeine around, but the good news is that all of the principals behind building a testable architecture make good software engineering sense as well. In other words, the more testable the architecture the better the resulting design and a better engineered system will be produced. Any suggestions that we make to the developers to enhance test capabilities will also have other positive effects that will help the developer. We are not asking the developer to do any extra work, solely for the purpose of testing. The best way that I've found to influence a developer to design the application in a certain way is not to explain why it's easier for test but to describe how it makes her life easier. It's also important to note that none of the suggestions for a testable architecture cost any more than building robust, well-engineered software.

OK, so you have convinced yourself you want a testable architecture, and you are ready to work with development to guide them towards building one. What specific design characteristics should you ask for? Here are my suggestions, feel free to add or modify these for your particular environment:

Instead of building a system that is one huge chunk of code; look for a system that is based on fairly independent components. These components should interact with each

other through well-defined interfaces. Further, the interfaces should be fully defined before any code on either side of the interface is written. Writing component-based software has several advantages. First, a large, complex problem is broken-down into several smaller, more manageable problems. This allows for simpler solutions to the simple business-logic problems and allows the developer to be creative where he needs to be, namely on solving the complex business problems. Second, components enable dividing work among several developers who can work relatively independently, which will bring the product to market faster. The less well-defined the work between developers is, the more time they will have to spend coordinating so they don't break each other's code. Third, the problem of system upgrades may be simplified, because designed correctly, components can be upgraded independently. Further, if one or more components are from a third party (not developed internally; whether purchased or open-source), it can be upgraded independently from the other components. Sometimes third-party components need to come from a different source than originally thought. For instance, let's say your legal department just found out that your "open-source" component that you shipped with a previous version of your software is not licensed for commercial applications or somebody found a lower-cost component. In either case it's likely to be dictated to the product team that the existing code needs to be taken out of the system and another one substituted. If you kept the use of this component through a well-defined interface, swapping it out is relatively easy. If the component's use was spread throughout the system it could be very time consuming to find all the places it is used, re-code and re-test. Fourth, when a component is enhanced, or a bug is fixed, or logic is changed for some business or performance reason, the likelihood that it affects the rest of the system is reduced. Unexpected side-effects, which is the bane of many product teams are more easily avoided, or if not completely avoidable, more detectable making its impact easier to manage

Now you are probably asking yourself, "OK, I see how to look for componentization, I see the advantage to developers, but what is the advantage to me, the tester? Is Jeff saying I need to test EVERY component by itself? Isn't that white-box testing?". Good questions. Let's continue. Just as components break-down the development problem into several, more manageable problems, it does the same for testing. Testing a smaller set of problems allows us to think of ways to more thoroughly test that component, allows us to divide the work among more than one test engineer, and divides in our mind, testing the features and functions of any given component from testing the system as whole. When these two issues (feature and system testing) are jumbled up it's harder for us to keep the distinctions clear in our head so that we can do a good job of each. And no, I am not saying to test every component and the component level. Pick out the "most important" ones for your system and test those. Examples of important components are: components that are exposed, via an API or a user interface to a customer or user. Components that contain system critical functions such as the "File->Save" command of a word-processing application or components where performance is critical. By "performance critical" components I am talking about those components where the timeliness of the data is as important as the accuracy of the data. Components that are protecting the application from its environment, such as resource shortage (CPU, disk, Memory), slow network, bad user input, or protecting security vulnerabilities. In other words,

components that many not be directly involved in business logic but should the code in them need to be exercised, you want to be 100% sure the expected results are achieved. Having a good, solid regression suite for these types of components is a comfort to developers because it gives them one more level of assurance that should they need to change or enhance one of these components, the fix can be checked for regression in isolation from the rest of the system.

By the way, I still consider testing at the component level “black-box” testing, because this type of testing is not requiring the tester to understand or examine the code itself. The test engineer does not need to look inside the component so it’s not a “white-box”. By the way, “white-box” testing should probably be called “clear-box”, because if it’s white, you still can’t see inside it.

While we are talking about components and the APIs that define the communication or “transition” between them, it’s good to take note that these areas of transition are excellent places to find bugs. By playing music in several bands over the years, I came to realize that the transitions (beginnings, endings, key changes, tempo changes, moving from chorus to verse and back again), are the most difficult parts to execute correctly. This analogy hold true between software components as well, since the tricky parts are the transitions, testing at these points is fertile ground for bug hunting.

Since you will now be testing at software by calling the API’s directly, one important objective is ask developers to specify their API signatures (method or routine name, incoming parameters, outgoing parameters, return values and side-effects), as early as possible in the development process. You may encounter resistance to this, because often the developer wishes or is under-pressure from management to “get something working” as quickly as possible. This means they might wish to define their APIs as they are implemented. This is bad for testers because, it stops them from designing or implementing detailed test cases, until the APIs are defined, or worse, until after the code as is handed off to test. Later delivery of API signatures can be a real obstacle to effective and timely test automation. Of course, delaying test design and automation is just one disadvantage to late delivery of the signatures. Other developers, partners and customers, in short any consumer of the component needs the API signatures as early as possible so they can build their piece of the software as well. Specifying API signatures before coding has another advantage, it forces the developers, possibly working in different teams or locations, to think about (and hopefully send out for review), what they are going to build before they build it. Just as a testers can look at a published API signature and think of test cases that will not only check functionality, but exercise error handling, stress testing, etc., other developers that are going to consume this API get a good feel for the capabilities that are being built into this component so they know how to build their own component. By specifying and reviewing these signatures the entire development team can agree to an external design before coding begins. Please keep in mind that even in the most ideal environment these signatures will change somewhat during development. It’s nearly impossible to think of everything up front, and external issues such as performance or unexpected external conditions will force some API changes during the coding phase. The important task for developers and testers alike is to try to

minimize these changes by careful consideration, review, prototyping etc. of the APIs. Early API specification is another example where paying attention to testability helps the overall project more than just enabling better testing.

The user interface (UI) whether graphical, command driven, voice activated, or using any other input technique should always be separate from the business or application logic. The UI component's only job should be taking input from and displaying output to the user. Any other work (input verification, business logic, error checking etc.) should be done by the back-end or engine. The UI and back-end should communicate through a well-defined and fully specified Application Programming Interface (API). This means that the UI will be its own component. Test can use the API between the UI component and the back-end logic to fully test the back-end of the product without invoking or going through the UI. There are several advantages to this. First, it may be tricky (or even impossible, to fully test the back-end by interfacing it solely through the UI. Second, when testing the entire system together if a bug occurs, it won't always be clear if it's an issue with the UI code or the business logic code itself, independent testing answers this question. Third, UI input is not as easy to encode, and UI output is not as easy to decode, as direct calls to the API. A test engineer can spend most of her energy working these encode/decode issues; energy that is far better spent writing those killer test-cases. Fourth, the user interface design is often fluid, controversial and subjective. UI changes very late in a product's lifecycle are very common and difficult to completely avoid. Having a strong separation between the front and back-end, allows changes to the front-end while eliminating or greatly reducing the effect on the back-end. Less work is wasted then, by both development and test.

As all of the testability sections previously stated, a solid separation between front and back-ends makes sense for both the development and test teams. Separating the front and back-ends not only reduces the impact to UI changes it provides a solid foundation for managing multiple UIs. Each UI interacts with the back-end re-using the same API, maximizing the code that is common. The UI component itself can concern itself with the specific problems of gather and presenting data, which is precisely what differs in the multiple UIs. When a product manager or another very creative person comes along with a new kind of UI, (voice activation, heads-up displays reading eye-movement, mind reading) much of the back-end work can be re-used.

Of course, test the components alone will not be a complete test of the system. At some point the entire system will should be tested together, but it's much easier to perform this testing if there is some degree of confidence that the UI and back-end have passed as many test cases as possible before putting them together.

Another aspect of testable design that I would like to touch on is log files. From either a testability or a troubleshooting point of view, it is advantageous to log as much activity as possible. This can be configured to be turned on or off as necessary because it's easy to really kill performance with all of the I/O involved in log file generation. In my opinion, however, you can't enable enough logging. It's far easier to figure out what a program is doing by reading log files than examining source code. It also gives an important tool to

users and technical support for visibility into the application when it's not behaving properly. One disadvantage to logging everything is it that makes analysis difficult for a human simply using only his eyes. Users and technical support often wish to filter logs so they can easily ignore records they don't care about. Test engineers want to programmatically analyze logs as a way of determining test pass or fail. For these reasons, it's important that the logs be written in an easily parse-able format. Field formats such as date and times should only appear one way. Field locations should be as regular as possible with the order varying as little as possible.

If you haven't figured it out yet, or already forgot the betting of the article, I want to remind you, that the time to start engaging the developers about a testable architecture is very early in the product lifecycle. Before functional specifications are written, discussions should occur on automation approaches and strategy. The earlier the test engineer's needs are communicated to the development team, the easier it is for development to keep these needs in mind and design to them as well as the product requirements.

In summary the important ideas here is that testable designs ease testability and enable more robust automation while simultaneously increasing the software quality. Designing for testability is a long-term effort that might take several releases of your product to perfect. Keeping the ideal in mind however, while making progress from one release to the next will produce desirable results and enable a feedback loop for continuous improvement.

Good luck and happy testing.