"The Living Creature" - Testing Web Applications

Table of Contents

Web Application Architecture

What do we mean by "Web Application"? There is an incredible range of sophistication in web applications from a simple company web site with "brochure ware" to sites like Yahoo or Amazon with complex search engines and order fulfillment. One way to look at the web application architecture is to take the model of a traditional business transaction application and to replace the user front end by the web site. A customer acquires goods and/or services from your company, in exchange for money. There are mechanisms in place to facilitate that transaction between client and company. Instead of a sales rep, a clerk, a cashier, or such person, you have a browser pointing at a web site. The company is never closed! Customers can serve themselves!

Think of a vending machine: this machine fills orders based on input from users, verifies transfer of funds, and has a basic user interface. Now add some complexity: make the UI a browser-based solution that must run in multiple browsers on multiple operating systems instead of a touch pad, and have the machine fill orders directly from a warehouse in the mid-Western US, while tracking and re-stocking (real-time) inventory. In general, people will not be putting coins in the machine, but entering their credit card numbers - which requires real time access to credit companies to have each transaction approved. Moreover, we would expect that all credit card information should be extremely secure.

The average web application architecture is shown in Figure 1 (below). The client end of the system is represented by a browser, which connects to the web site server via the Internet. The centerpiece of all web applications is a relational database which stores dynamic contents. A transaction server controls the interactions between the database and other servers (often called "application servers"). Fulfillment may include interfacing with financial institutions, warehouse systems, etc. The administration function handles data updates and database administration. Of course, there are many possible permutations that form this basic picture.

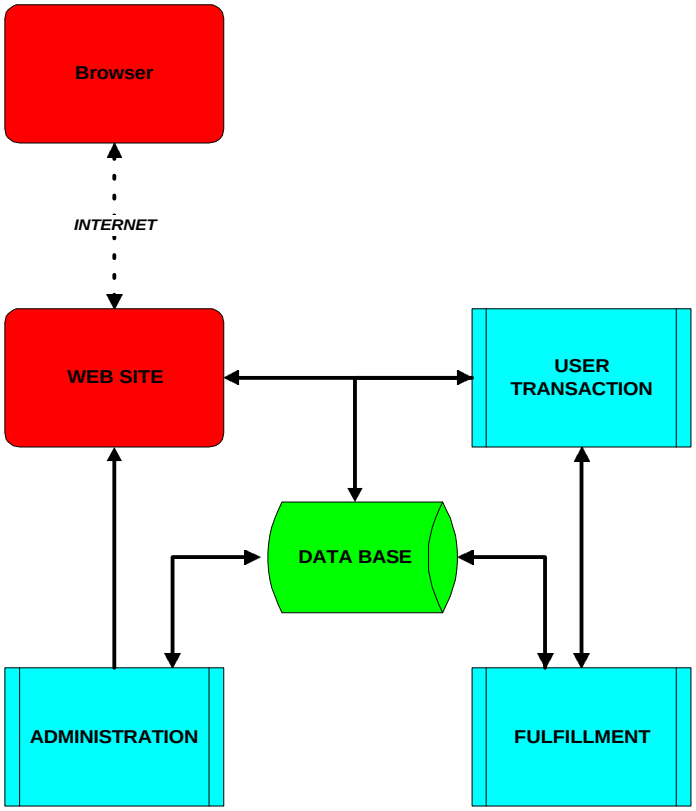

Figure 1: Web Application Architecture

Considering this architecture, it should now become clear that web applications are not simply web sites with some artwork and some HTML or Java. They are very similar to the traditional transaction systems with additional complexity at the front end. The testing effort required for such a system is considerably larger than for applications without web interface.

The Development Life Cycle of a Web Application

Most of us have been exposed to a few software development lifecycle models, such as the Spiral model, the Waterfall model, and so forth. While the typical software project includes such phases as planning, requirement gathering, analysis and design,

implementation (coding), integration, testing, release and maintenance, how do these phases match up for a web application project? Here is one view the authors have frequently observed:

Typical software project	**Web application project**
Gathering of market/user requirements *"What are we going to build? How does it compare to products currently available?"* This is typically supported by a detailed requirements specification.	Gathering of market/user requirements *"What services are we going to offer our visitors/customers? What is the best user interface and navigation to reach the most important pages with a minimum of clicks? What are the current trends and hot technologies?"* This is typically based on discussions, notes and ideas.
Planning *"How long will it take our available resources to build this product? How will we test this product?"* Typically involves experience-based estimations and planning.	Planning *"We need to get this out NOW! Marketing has picked this date to go live; we'll just have to have it done by then (typically 3 to 4 months)."* Purely driven by available time window and resources.
Analysis and Design *"What technologies should we use? Any design patterns we should follow? What kind of architecture will allow us to reuse code most effectively, particularly for future versions?"* Mostly based on well known technologies and design methods. Generally complete before implementation starts.	Analysis and Design *"How should the site look? What kinds of logos and graphics will we use? How do we develop a 'brand' for our site? Who is our 'typical' customer? How can we make it usable? What technologies will we use?"* Short, iterative cycles of design in parallel with implementation activities.
Implementation *"Let's decide on the sequence of building blocks that will optimize our integration of a series of builds"* Sequential development of design components.	Implementation *"Let's put in the framework and hang some of the key features off of it. Then we can show it as a demo or pilot site to our prospective users."* Iterative prototyping and story-boarding with gradual transition of a prototype to a production site.

Integration *"How does the product begin to take shape, as the constituent pieces are bolted together? Are we meeting our requirements? Are we creating what we set out to create in the first place?"* Assembly of components to build the specified system.	Integration This phase typically does not exist. It is a point in time when prototyping stops and the site goes live.
Testing *"Have we tested the product in a reproducible and consistent manner? Have we achieved complete test coverage? Have all serious defects been resolved in some manner?"* Systematic testing of functionality against specifications.	Testing *"It's just a website -- the designer will test it as (s)he develops it, right? How do you test a website? Make sure the links all work?"* Testing of implied features based on a general idea of desired functionality.
Release *"Have we met our acceptance criteria? Is the product stable? Has QA authorized the product for release? Have we implemented version control methods to ensure we can always retrieve the source code for this release?"* Building a release candidate and burning it to CD.	Release *"Go live NOW! We can always add the rest of the features later!"* Transfer of the development site to the live server.
Maintenance *"What features can we add for a future release? What bug fixes? How do we deal with defects reported by the end-user?"* Periodic updates based on feature enhancements and user feedback.	Maintenance *"We just publish new stuff when it's ready…we can make changes on the fly, since there's no installation required. Any changes should be transparent to our users…"* Integral part of the extended development life cycle for web apps.
Average timeframe for the above: **One to three years**	Average timeframe for the above: **4 months**

There are many possible terms for the web app development life cycle including the spiral life cycle or some form of iterative life cycle. A more cynical way to describe the most commonly observed approach is to describe it as the unstructured development similar to the early days of software development before software engineering techniques were introduced. The “maintenance phase” often fills the role of adding missed features and fixing problems.

The fundamental difference from traditional development is the extreme time pressure under which web systems are developed. This necessitates a new approach, which can severely limit the scope of the initial release and pushes many features into iterations that follow the initial release. The resulting life cycle model could be called a whale/dolphin model. The whale is the initial hump of effort to release the first version of the web app and the dolphins are subsequent cycles of functionality increments. These subsequent cycles continue throughout the life of the web application (see Figure 2 below).

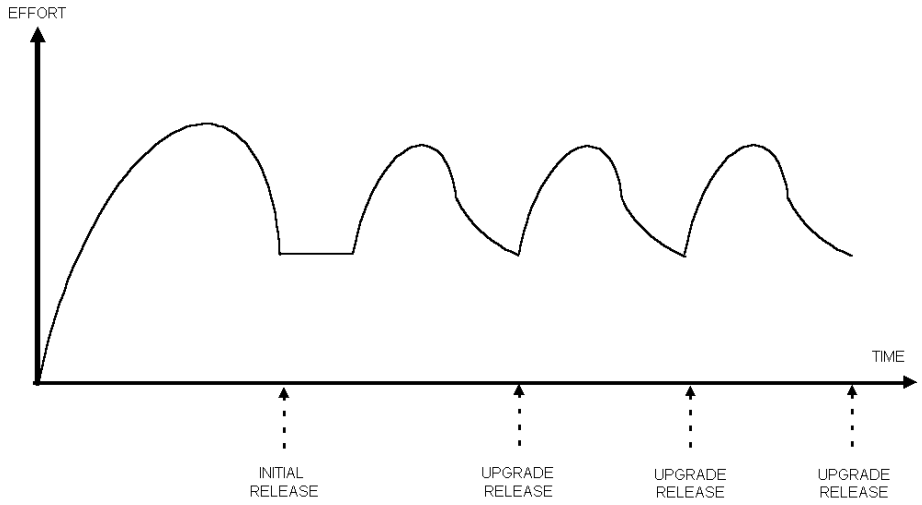

Figure 2: Effort vs. Time for a Typical Web Application

The first step in planning life cycle activities for a web application is to recognize the iterative nature of this product life cycle. Planning for subsequent releases right at the beginning of the project will greatly facilitate tradeoffs that may have to be made in each of the cycles.

Requirements

Requirements are traditionally the foundation for test planning. Web application projects often have requirements that are fuzzy to start with, subject to rapid changes throughout the project lifecycle, and often have more to do with marketing, art, branding, and advertising than providing a solid set of core functionality that works. Typically, there is an initial vision for the basic functions and the look-and-feel for the site. This vision is implemented for the initial release. However, the initial vision is subject to frequent updates and additions; especially once the initial release has gone live. A major reason for the initial release is to meet time-to-market requirements while postponing additional features that often are added later. As a result, we have an initial development lifecycle followed by ripples of smaller cycles for feature additions. The web-time that so constrains these kinds of projects usually does not allow for complete documentation and analysis of the initial requirements and this situation does not improve much for future additions.

This makes test planning difficult. It may have to be based on intuitive interpretation of intended functionality. Even more aggravating is the very fluid nature of the functionality of the site design and the lack of documenting this change. As a result, many test efforts may go off into a “wild goose chase” testing for functionality that was changed or deleted.

> "We'll write them as we go."
> "We don't have time to write requirements.... We have to ship!"
> "Requirements? For a website?"

Sound familiar? The following is a direct quote from *The Mythical Man-Month* by Frederick Brooks, Jr.: "The most pernicious and subtle bugs are system bugs arising from mismatched assumptions made by the authors of the various components" (p.142). The writing of requirements is one simple method to reduce the amount of assumptions that have to be made by project personnel, including testers. Application software developers, testers, and managers have learned, from painful and expensive mistakes, that this approach can make all the difference in delivering a solid product to the customer.

Serious testing is impossible in an environment of uncontrolled changes to requirements. Combining this statement with the realities of web development, we suggest a simple technique that has proven useful for both the development team and the test group. Requirements and functional specifications for the initial release should be captured in a document. Any subsequent changes (called change requests or "CRs") should be managed and subjected to rigorous scope management. The set of test cases will then be based on the combination of the initial spec and the approved CRs. In the end, it is not uncommon that the set of approved CRs will represent the majority of the feature set for the system. There are many change request management systems on the market that can be used for this purpose.

Implementation: The Prototype as Product

One of the fundamental software engineering principles is that prototypes are thrown away, never shipped. Occasionally, a company will, under time-to-market pressures, ship a product that has morphed from a prototype, often with poor results. However, it is common practice to ship web applications that either are prototypes themselves or are early descendants of prototypes. The implementation phase then looks much like *evolutionary development*, where requirements changes lead to an evolutionary sequence of prototypes. Any of these prototypes may be declared to be the initial production version when time for additional iterations runs out.

This method bears a high risk of not allowing for sufficient testing before the site is launched. If the development organization feels that it must follow the evolutionary model, each iteration should be finished off with a test pass on a separate staging server that is not subjected to any interference from development changes.

Quality in the Web-Space

Remember, many different companies define quality in many ways, but to the end user, quality always means, "Am I satisfied?" If we define quality as perceived value to the user over cost to the user, then web applications, which have no real cost to the user, should move towards infinitely greater quality. In fact, as the cost goes down and the marketplace becomes saturated with more and more sites that offer similar transaction functionality, and as these sites become increasingly complex, we see the opposite occur. Decreasing quality of software, for both desktop applications and web applications, is something that has been occurring for quite some time now, and the sad truth is that end users have been tolerating (if not encouraging) this trend.

We believe that market forces will cause this trend to reverse. As the Internet becomes swamped with dot.coms competing against each other, the limited attention span of users will reward only those sites that do not disappoint the user. This is the key argument for increasing pressure towards better testing of web applications: with traditional software a user has spent a certain amount of money and hence feels motivated to get the best utility from his or her investment. The alternative would only be to buy a different solution (which implies more expenditure, a new learning curve, incompatibilities etc.). A web user does not have such a difficult choice. If she uses an airline reservation system that does not behave according to expectations, she can click through to another service provider and see if she likes that environment better. The switching costs for most web applications are so low, that users will simply browse away from sites that exhibit poor usability. (Refer to Jakob Nielsen's and Donald A. Norman's article for *InformationWeek Online* entitled "Usability On The Web Isn't A Luxury" at http://www.informationweek.com/773/web.htm, or refer to Dr. Neilsen's website, http://www.useit.com, for more information on usability in general.) We believe that quality and functionality of web sites will ultimately be a major factor in the inevitable shake-out, which will reduce the number of redundant dot.coms.

Some Quality Factors of Web Applications

Security, reliability, and recoverability are all issues that can make or break a site. Up-time requirements for web applications are far more stringent than for off-the-shelf/shrink-wrap software. Web sites are just **not** allowed to fail, become corrupt, or exhibit poor usability, while the average computer user may tolerate buggy software (but not so buggy that they cannot do their work).

Some key quality factors that can be related to web applications include:

1) Reliability;
2) Recoverability;
3) Security;
4) Usability; and
5) Performance.

Reliability
One definition of "reliable" is *exhibiting a reasonable consistency in results obtained*. How many web sites today can be called "reliable"? Another definition is *that which may be relied upon, worthy of confidence, trustworthy*. Perhaps a web application is trusted by users who use an on-line banking web application (service) to complete all of their banking transactions. One would hope that the results are consistent. However, perhaps the site is not always accessible on a consistent basis or displays periodic performance problems.

One example is that of a financial institution which set-up on-line trading services for its clients. Many customers signed up for and used the service. However, this institution did not grow the back-end application and database servers *and* the fulfillment personnel (brokers, traders, etc.) in line with the growth of the customer base. It wasn't long before customers could not access the site for periods of three to fours hours at a time, and when they did get access to request a trade, it often took several additional hours before their trades were processed. One can easily image the frustration and potential financial losses of the customers when their trades could not be completed on time and as requested. This would be an example where a lack of consistency implies a lack of reliability. (This scenario, from the perspective of the developing company, may be more of a scalability issue; however, from the user's point of view, it is a reliability issue.) A simpler example of reliability might be a shopping site that is constantly unavailable to its users, whether due to excessive traffic or hardware/software problems. A user will, after successive failed attempts to connect to the site, browse to a competitor's site.

It is the opinion of the authors that reliability of web applications is purely a user-based quality factor. That is, the definition of "reliable" as it applies to web applications is a subjective one, and that the user is the creator and modifier of such a definition. Having web application project personnel define "reliability" as it should apply to their web application is misleading. This is one more reason to acquire information regarding targeted audiences when designing and testing web applications.

Recoverability
This is another quality factor that is often ignored or put-off until after the initial release. Many web applications will have a back-up or "redundant" server -- a server to which web traffic is rerouted should the primary server fail. This set-up may be mimicked for the database server component(s) as well. The re-routing mechanism(s) must be tested with methods similar to tests performed on fault tolerant systems. However, recoverability implies much more than a fail-safe switchover. It has to be re-synchronized with all connected systems, such as warehousing systems, payment fulfillment operations, etc., as well as performing data validation to ensure that data has not been lost or become corrupted. This can increase the complexity of test scenarios significantly.

The potential financial losses from having your web application unavailable are large. If a user finds your service unavailable for an excessive period of time (excessive from the user's perspective), the likelihood of that user switching or browsing to a competitor's service is increased. The quicker your site can recover in a manner that is transparent to

the user, the less chance you give that user to browse away from your site, and thus your services. If the site cannot recover quickly (or in a manner that is transparent to the user), the next step is to manage the user's expectations, by informing the user when you expect your site to be available and functional. Explaining to a user that your site will be available within 24 hours from the shut down will allow you to keep more users coming back to your site once restored, as compared to providing the user with no information.

Security

Probably the most critical criterion for a web application is that of security. The need to regulate access to information, to verify user identities, and to encrypt confidential information is of paramount importance. Credit card information, medical information, financial information, and corporate information must all be protected from persons ranging from the casual visitor to the determined cracker. There are many layers of security, from password-based security to digital certificates, each of which has its pros and cons. For a good on-line reference about security issues, refer to the W3C FAQ on Security (http://www.w3.org/Security/Faq/www-security-faq.html).

Many of the security measures used for web applications are third-party products. Certificates, Secure Sockets Layer (SSL) software, web server software (IIS, Apache, etc.), and so forth, are all products created by companies other than your own. They all have defects, some of which can be exploited by those who wish to intercept secure data, corrupt your servers or databases, hijack your content and/or scripts, or tap into password files. When a security defect is discovered, it can difficult to see a solution in a reasonable time frame from a third-party vendor. Which leaves only one feasible option: modify *your* product to deal with these defects.

Some methods of reducing these kinds of risks include:

1) Researching current security issues in third-party products you plan on using - this may affect your decision to use that product;
2) Reviewing your design to try to remove as many potential security issues as possible, including examining architectural designs;
3) Writing "defensive" code, as demonstrated with the rules for good defensive Java programming listed in Chapter 7 of Securing Java, by Gary McGraw and Edward Felten, (copyright 1999, John Wiley and Sons) available on-line at http://www.securingjava.com/chapter-seven/chapter-seven-1.html.
4) Creating coding standards that reflect some of these "defensive" techniques; and
5) Performing code inspections on the high-risk components of the web application.

A dangerous and common approach is to release software and wait for someone to discover a security-related defect, which can then be addressed by releasing a "patch." For a web application, the patch can usually be installed in a seamless manner (one that is transparent to the user). However, since security has a large dependence on environment, changing the environment by installing such a patch may put your web application at risk.

Usability
Usability is a critical area for a web application. In the past, only very sophisticated projects consulted GUI designers or experts in human-machine interaction. These specialists are now in greater demand since the success of a web application depends largely on usability. Many of the web GUI designers are in fact artists, desktop publishers, and the like. They may not have had formal training or even exposure into some of the basic GUI and usability principles.

Usability of web applications will gain in importance over the next few years, as more and more web applications emerge, and as web application users become more experienced, or “sophisticated” users. The sophisticated user will have very different usability issues than a novice user. To date, the Internet has been about providing information (largely textual information) and a few services to relatively inexperienced users. As that model moves towards increased services for intermediate or sophisticated users, new approaches will have to be undertaken and implemented.

How do you test this without a large beta test group? Some companies are using the concept of a "pilot" site, which is a scaled-back version of the web application, accessible only to a small to medium-sized focus group. Feedback regarding the interfaces, the look and feel, and the workflow of the site is then collected from these groups and used to modify the application before it goes live (or for future iterative releases). Understanding the types of users that will visit the site will allow initial design efforts to focus on those kinds of users. The focus group can then be used to confirm or correct the initial design.

There are two approaches to this problem. The first is to capture and quantify the meaning of learnability, understandability, and operability in a testable form. In other words, it is an attempt to formulate these testable requirements by describing how they are supposed to be accomplished. This may then form the basis of a usability test plan. A second approach is to gather a group of representatives of the target user community and to let them work through the site while observing their problems and the bugs they run into. However, this approach may make it difficult to clearly determine what constitutes a bug. There are many techniques that can be used to help distinguish usability defects from feature requests, such as using "thinking aloud" and "question asking" combined with basic performance information gathering. (For basic information on these techniques and reference links to other resources, refer to James Hon's *The Usability Methods Toolbox* website at http://www.best.com/~jthom/usability/.)

Performance
Performance testing involves testing a program for timely responses. The time needed to complete an action is usually benchmarked, or compared, against either the time to perform a similar action in a previous version of the same program or against the time to perform the identical action in a similar program. For example, the time to open a new file in one application would be compared against the time to open a new file in previous versions of that same application, as well as the time to open a new file in the competing application. The benchmark time for a piece of software can also be defined explicitly by

the client in the requirement specification document. When documenting these requirements, state the performance requirements in terms of real numbers instead of using statements like "must be at least as fast as the previous version." Using real numbers makes reporting performance problems much easier to rank in order of priority and severity. Report performance problems in terms of the percentage of improvement needed to meet the performance requirements.

When conducting performance testing, be sure to pay attention to the data volume (i.e. file size) so you're comparing apples to apples instead of apples to oranges. Calibrate your performance test machine against other machines of the same class to make sure that the times are comparable. Make sure that at least some performance testing is done early, even if the formal performance testing is scheduled to be conducted at a later milestone. Early performance testing will give an indication of how much work will need to be done in order to meet the performance requirements. Conduct some of the early performance testing on a minimum configuration machine. Include connecting via a dial-up connection with a modem (many users will have connection speeds of 28.8k, yet many testers are testing via a T1 or xDSL connection) so any problems with performance can be identified as early as possible. Whenever possible automate performance testing so that it is more accurate and can be run on a regular basis.

One flavor of performance testing is load testing. Load testing for a web application can be thought of as multi-user performance testing, where you want to test for performance slow-downs that occur as additional users use the application. The key difference in conducting performance testing of a web application versus a desktop application is that the web application has many physical points where slow-downs can occur. The bottlenecks may be at the web server, the application server, or at the database server, and pinpointing their root causes can be extremely difficult. Refer to Mark D. Anderson's article in *Software Testing and Quality Engineering* (September/October 1999, Vol 1, Issue 5, pages 30-41) for a good discussion on load (multi-user performance) testing.

Testing in the Web-Space- What's different?

By now it should be apparent that testing web applications is not trivial. Testing a web page with relatively static content and little to no forms will take very little time (Are all the links correct and working? Does all content load correctly? Is loading time fast enough?). Testing complete e-commerce applications requires much more sophisticated testing strategies, and thus more time.

One of the major weaknesses of web application testing is inadequate technical expertise and ability. Testers have to understand subtle browser, operating system, web server, and database differences. The more they know about **scripting** (ASP, XML, HTML, etc.), **databases** (Oracle, SQL, etc.), **web servers** (IIS, Apache, etc.), and the **data transfer mechanism** behind the UI (TCP/IP, HTTP, FTP, etc.), the more effective they will be. Testers simply cannot just test the functionality by exercising the UI (in this case, the browser); they will miss all the other aspects of testing required for web applications

(such as performance, security, database integrity, etc.). Remember, crackers do not use browsers to crack sites; they use scripts.

As well, the lack of mature test tools makes automation difficult. This situation is reminiscent of when Java first hit the high-tech scene. Developers and project managers alike wanted to use this new technology. Testers suddenly had their workloads doubled, tripled, or more, simply because of the number of configurations and the lack of any mature testing and automation tools. One company for whom one of the authors worked, had developed a large Java application, and was forced to write custom debugging and automation tools because, at the time, nothing comparable existed. This situation will not change as long as web technology continues to evolve at the current rate.

The Test Environment - Does such a thing currently exist in web-space?

Poorly defined development and test environments can hamper version control and other configuration management efforts. How do you rollback code changes when you have no previous build? How does new functionality and defect fixes get migrated into each build? Does the term “build” mean anything in the web-space? For most web application projects, it doesn't. Test personnel cannot revert to a "known state" if the source code is not being archived or not being labeled or branched in the version control repository. Not having a previous release to revert to for testing purposes makes isolating and analyzing defects more difficult as the environment continually becomes more complex. Another problem area is the common (and dangerous) practice of migrating defect fixes and new functionality to a live server prior to testing, and testing on live servers. Your test team shouldn't bring down your site; they should bring down a separate test server.

What is the best environment to set-up for a web-testing team? How can companies organize their environment to help testing, instead of making it nearly impossible? We suggest at a minimum three separate servers:

- A development server used for the development team. This may be a place for a prototype or simply to play with features.
- A staging server that is periodically updated with a new release. This server can also be used as the test platform. Crashing the staging server does not affect the development team or the live site.
- A production server that hosts the live site. Nobody, except an appointed web master is allowed to modify this server. Its contents are updated periodically from the staging server.

This is a minimum configuration and there are many variations of this basic scheme, especially when the full (final) web application itself consists of several servers, not all of which are necessarily at the same geographic location.

A test environment should mimic the destined (planned) deployment environment as closely as possible. Testing a web application in a LAN setting will not be the same as testing a web application hosted on a series of external web and application servers. Proper security testing cannot be performed on a web application in any configuration other than the final one. The complexity of the networking, security (firewalls, proxy

servers, etc.), hardware, and software configurations cannot be reproduced to any degree of accuracy in an internal LAN-type environment. If your company were developing software for hand-held devices, you would test on the intended devices, not on your NT machine. Yet, many companies choose to test in this manner. With a short test schedule, there can be some benefit to testing on an environment other than the final one - if the testing performed is only usability testing or beta group/pilot site testing with the sole purpose of gathering user feedback for future iterations.

Platforms and Browsers

Testing issues are aggravated by the large variety of browser and operating system compatibility testing required. Creating a matrix of operating systems vs. browser versions, we see how large this can be. Since it is very difficult to force users to use the latest browsers (there will always be someone using Netscape 2.x somewhere), you will have to test with older versions.

One cannot assume that browsers behave in similar ways. There are known differences between browser versions and between MS Explorer and Netscape. Nonetheless, it is important to understand the issue of multiple configuration testing, and how that can really grow your test needs in a hurry. Look at this basic matrix. Imagine having to test on each of these platforms with each of the main browsers!

	Netscape	Internet Explorer	HotJava	Other
Windows 95				
Windows 98				
Windows 98SE				
Windows 98ME				
Windows NT 4.0 Workstation w/SP 3				
Windows NT 4.0 Workstation w/SP 5				
Windows NT 4.0 Workstation w/SP 6				
Windows NT 4.0 Server w/SP 6				
Windows 2000 Pro				
Windows 2000 Server				
MacOS 9.0				
MacOS 8.6				
MacOS 8.0				
MacOS 7.6				

This list does not even include WebTV, Linux, UNIX, and UNIX-like machines, nor does it express the multitudes of browser versions. There are significant HTML and/or XML compliance, security, and performance differences between browsers and between browser versions, so it is important to test on a variety of browser versions. A good understanding of the make up of the application's typical users and a breakdown of the

browsers and browser versions currently being used should help focus your test effort in this area. For example, Microsoft Explorer offers good performance when connecting to the IIS server while the Netscape Navigator browser performance is remarkably more sluggish in that configuration.

The goal is to separate the various configurations into two groups: a core set of "primary" configurations, on which the majority of your testing will be completed; and the remaining or "secondary" configurations, on which security, performance, functionality, system, and acceptance testing will be performed (time permitting of course). Separating the possible configurations into these two lists will allow a test team to best utilize the short amount of time allocated to testing.

Focusing Your Testing Efforts

Testing for usability, browser compatibility and other front-end aspects may easily detract from testing the functionality of the back-end. Fortunately, back-end applications represent more traditional software applications such as data base access, transaction servers, etc. In many cases, these functional areas can be isolated and exercised using test drivers and stubs. Provided the system is designed using a modular architecture, testing the back-end components can be done in parallel to the development and testing of front-end components.

Critical for overall test planning is to recognize that there will probably not be enough time to test all parts of the system exhaustively. In an ideal world, we would develop a test plan and systematically work through the plan. In web time, this is not likely to happen. Here is what we suggest:

- Develop a test plan which is as complete as possible
- Structure the plan by functional component
- Prioritize components using the associated risk of failure as a guide to determine the priority
- Start testing the high priority items and work your way down to lower priorities until you run out of time.

The required risk prioritization is the most critical part. It should be derived from the business case, reviewed with those who define the product vision, and verified with user representatives.

Conclusion

Web applications require a new perspective on software development and testing practices. While many activities are similar to traditional software engineering, we have to adapt to the new realities of developing in web time. Some of these realities, including shorter life cycles, multidisciplinary teams, rapidly changing technology, increase the risks of development. In the past, we learned to increase the rigor of the development process to manage increased risk. However, in an era of shorter and shorter development cycles, the time available for process must also be shortened. The only way out of this conundrum is to focus on those process aspects that give the highest return for the time

invested. Each team member has to work smarter, not just harder. When it comes to testing, this means that the time spent on test planning is becoming even more valuable since it allows us to test strategically important parts of the application instead of testing tactically convenient ones.